UiO : Department of Informatics
University of Oslo

Improving Dissemination Reliability of P2P Topic-Based Publish-Subscribe Systems Exploiting Node Regularity

Endri Hysenaj

Master's Thesis Spring 2014

Improving Dissemination Reliability of P2P Topic-Based Publish-Subscribe Systems Exploiting Node Regularity

Endri Hysenaj

25th February 2014

Abstract

Publish-subscribe paradigm of communication is one of the most popular and powerful models for services on the Web. Given the increase in popularity of P2P systems and the benefits of them over centralized versions,researchers have created interest in making P2P publish-subscribe systems. However, P2P systems' characteristic property is that nodes in them come and go from the system as they please, forming churn. Recent studies of P2P networks have found that nodes tend to have recurring patterns of availability which are regular over time. There have also been studies for taking advantage of this regular behavior of the nodes in distributed storage systems. The purpose of this thesis is to research into the possibility of using the regularity in node availability for P2P topic based publish-subscribe systems. We explore more into detail this property of the nodes in order to determine which parameters affect the perception of nodes as being regular. We create a generic service called GRID that identifies a node's own regularity pattern and discovers other regular nodes in the system for a specific timeslot. We later make an application that takes advantage of GRID to prove that this property of the nodes might be used to increase the dissemination reliability and speed of an existing P2P topic-based publish subscribe system called PolderCast. We show that, while introducing a small amount of unfairness and higher number of connections on the nodes, the overall dissemination is improved, especially in cases of high churn.

Acknowledgments

First of all, I want to thank my family for making my studies at UiO possible and for all the love and support they have given me during this period.

Great thanks go to my supervisor Roman Vitenberg for his helpful guidance and support. The discussions with you and your feedback have been most instructive.

Special thanks go to PhD candidate Vinay Setty for providing the source code of his work on PolderCast. Thank you for all the help and for your input in the discussions and feedback on my work.

I would also like to thank all of the teachers and students I have had the privilege and pleasure of interacting with during my master studies.

Last, but not least, I would like to thank my fiancé, Orjana, for all the patience, understanding and encouragement she has given me all this time. Thank you for all of your support.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
  - 1.1.1 Popularity of Publish-Subscribe . . . 1
  - 1.1.2 P2P Publish-Subscribe . . . 3
- 1.2 Problem Statement . . . 4
  - 1.2.1 Ideal P2P Topic-Based Publish-Subscribe . . . 4
  - 1.2.2 The effects of churn on different overlays . . . 6
  - 1.2.3 On-line node behavior . . . 6
- 1.3 Goals . . . 9
- 1.4 Contributions . . . 10
- 1.5 Assumptions . . . 10
- 1.6 Document Organization . . . 11

2 Background **13**
- 2.1 Categories of Publish-Subscribe . . . 13
- 2.2 Architectures of Publish-Subscribe Systems . . . 15
  - 2.2.1 Centralized architectures . . . 15
  - 2.2.2 Distributed Event Broker Network . . . 16
  - 2.2.3 Peer-to-peer Publish-Subscribe . . . 16
- 2.3 What is regularity? . . . 20
- 2.4 Gossiping and dissemination protocols used in this thesis . . . 22
  - 2.4.1 Cyclon . . . 23
  - 2.4.2 Vicinity . . . 24
  - 2.4.3 Rings . . . 26
  - 2.4.4 Dissemination protocol . . . 28
- 2.5 Evaluation metrics . . . 29

3 Related Work **31**
- 3.1 P2P overlays for topic-based publish-subscribe . . . 31
- 3.2 Churn Analysis . . . 33
- 3.3 Using statistical information about nodes' availability . . . 34
  - 3.3.1 Representation, exchange and on-line maintenance of node behavior information . . . 34
  - 3.3.2 Putting availability/regularity information to use . . . 36
- 3.4 Conclusions . . . 37

4 Regularity in existing systems **39**
4.1 Choice of traces for the analysis . . . 39
4.2 Description of the traces . . . 40
4.2.1 AVT - Availability trace format . . . 40
4.2.2 Skype trace . . . 40
4.2.3 KAD trace . . . 41
4.3 Methodology of analysis . . . 42
4.4 Results of trace analysis . . . 43
4.4.1 Relationship between timeslot length and number of regular nodes . . . 44
4.4.2 Relationship between regularity threshold γ and the number of regular nodes . . . 47
4.4.3 Percentage of nodes retaining regularity between timeslots . . . 49
4.5 Conclusions . . . 51

5 GRID: A generic regular nodes discovery service **53**
5.1 Motivation: The need for a generic service . . . 53
5.2 Functionality: Goals of the service . . . 53
5.3 Design . . . 54
5.4 Interface to the service . . . 58
5.5 Implementation . . . 59
5.5.1 Own Regularity identification service . . . 59
5.5.2 Peer Sampling service . . . 60
5.5.3 Victor: Generic clustering service by extending Vicinity 60
5.6 Limitations . . . 66

6 Taking Advantage of Node Regularity in PolderCast **69**
6.1 Introduction . . . 69
6.2 The desired overlay and dissemination . . . 70
6.3 The protocol structure . . . 71
6.4 Algorithms and implementation details . . . 73
6.4.1 Node Descriptor . . . 73
6.4.2 Use of GRID . . . 73
6.4.3 RingsRegular . . . 77
6.4.4 Dissemination Algorithm . . . 79
6.5 Possible challenge: Dealing with not synchronized clocks . . 87
6.6 Experiments' setup . . . 87
6.6.1 PeerSim . . . 87
6.6.2 Datasets and Availability Traces Used . . . 88
6.6.3 Parameters Used . . . 89
6.6.4 Metrics to be observed . . . 90
6.6.5 Baseline and simulations with regularity . . . 91
6.7 Experimental results . . . 92
6.7.1 Hit Ratio . . . 92
6.7.2 Node Degree . . . 96
6.7.3 Number of messages sent and received . . . 99
6.7.4 Path length . . . 102

6.7.5 Message Redundancy 102
6.7.6 Possible improvements to evaluation in the future . . 103
6.8 Conclusions 105

7 Summary and Conclusions **107**

8 Future Work **109**

List of Figures

1.1 A simple overlay demonstrating the disadvantage that arrives if node B, which will go off-line in 30 min, is chosen as a neighbor, versus equivalent node D which will be on-line (most probably) for several hours. 7

2.1 Visual conceptual representation of a topic-based publish-subscribe system 14
2.2 Visual conceptual representation of a content-based publish-subscribe system 14
2.3 Centralized architecture with one server 15
2.4 Centralized architecture with network of servers 16
2.5 Example of how the concept of CRB and timeslot fits into the flow of time 21
2.6 Structure of PolderCast's three layered approach 23
2.7 Rings overlay for one topic. 28
2.8 Dissemination over the Rings overlay for one topic 29

4.1 Presence of nodes in both systems 44
4.2 KAD daily regularity analysis with regularity threshold $\gamma =$ 70% with different timeslot durations 45
4.3 Skype daily regularity analysis with regularity threshold $\gamma = 70\%$ with different timeslot durations 45
4.4 KAD weekly regularity analysis with regularity threshold $\gamma = 70\%$ with different timeslot durations 46
4.5 Skype weekly regularity analysis with regularity threshold $\gamma = 70\%$ with different timeslot durations 46
4.6 Skype daily regularity analysis with timeslot duration of 60 min and different regularity thresholds 47
4.7 Skype weekly regularity analysis with timeslot duration of 60 min and different regularity thresholds 48
4.8 KAD daily regularity analysis with timeslot duration of 60 min and different regularity thresholds 48
4.9 KAD weekly regularity analysis with timeslot duration of 60 min and different regularity thresholds 48

5.1 The diagram of the architecture and interactions of the service with the application and between instances of the service on different nodes 55

5.2 An example of anti-correlation as presented in [22] 67

6.1 The desired topology of the ring for one of the topics once it is converged 71
6.2 The structure of the protocol stack and how the regularity related protocols gossip with each other 72
6.3 A dissemination scenario for one topic 80
6.4 CCDF of number of topics per node and number of nodes per topic in the Twitter subset of 1000 nodes we are using . . 88
6.5 Hit-ratio improvement when using PolderCast with F=4 versus PolderCast with F=2 93
6.6 Hit-ratio improvement when using regularity with $\gamma = 75\%$, medium churn speed 94
6.7 Hit-ratio improvement when using regularity with $\gamma = 70\%$, medium churn speed 94
6.8 Hit-ratio improvement when using regularity with $\gamma = 75\%$, high churn speed 95
6.9 Hit-ratio improvement when using regularity with $\gamma = 70\%$, high churn speed 95
6.10 Node in and out degrees averages time series, compared with PolderCast 97
6.11 Plot showing relationship between topic subscription size and in-degree of regular and normal nodes. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed. 97
6.12 Plot showing relationship between topic subscription size and out-degree of regular and normal nodes. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed. 98
6.13 CDFs of in and out degrees in regularity runs, compared also with PolderCast. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed. 98
6.14 CDFs of received and sent pub. msgs. in regularity runs, compared also with PolderCast. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed. 99
6.15 Topic subscription size and publishing messages received relationship. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed. 100
6.16 Topic subscription size and publishing messages sent relationship. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed. 100
6.17 CDFs of received and sent gossiping messages in Victor. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed. 101
6.18 CDFs of received and sent gossiping messages in RingsRegular. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed. 101
6.19 Number of Hops CDF of all messages sent. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed. 102

6.20 Redundancy CDF - All Messages. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed. 103

List of Tables

4.1 Average number of on-line nodes 43
4.2 Average Number of regular nodes for Skype for different regularity thresholds γ 49
4.3 Average Number of regular nodes for KAD for different regularity thresholds γ 49
4.4 Daily CRB, percentage of nodes that retain regularity between timeslots 50
4.5 Weekly CRB, percentage of nodes that retain regularity between timeslots 51

6.1 Table of notations used throughout algorithms. 74
6.2 Absolute hit ratio percentage difference between different protocols 96
6.3 Table of all other parameters used to benchmark the new protocol 104

List of Data Structures

1 Data structure of a *descriptor* of a node 56
2 Data structure for the view of Victor 61
3 Data structure of a *descriptor* of a node, extended for RingsRegular 75
4 The structure of the view of the RingsRegular protocol, $V_{RingsRegular}$. 78

List of Algorithms

1 Algorithm of the Vicinity protocol as presented in [46] 25
2 Algorithm for the Rings protocol, presented in [49] and [36] . 27

3 nextCycle algorithm for Victor 61
4 respondToRequest(Descriptor q, Descriptor[] receivedList) . . 63
5 *Descriptor*[] selectBestNeighbors(Descriptor referenceNode, Descriptor[] neighborList, Descriptor[] excludeList, int numToChoose) 65
6 int compare(Descriptor reference, Descriptor a, Descriptor b) 66

7 utilityScore(Descriptor a, Descriptor b) 76
8 Next cycle for RingsRegular protocol that handles the gossiping cycle of the RingRegular layer. 81
9 int[][] getUpdatedTopicCoverage() 82
10 void considerNodes(Descriptor[] nodeList) 82
11 applyHashFunction(Descriptor candidate, Descriptor[] neighbors, Descriptor selfDescriptor) 83
12 Descriptor[] selectBestNeighborsToSend(Descriptor d, Topic t) 84
13 Algorithm what to do when receiving a gossip response . . . 85
14 replyToGossipRequest(Descriptor q, Descriptor-Set received) 85
15 long getClockwiseDistance(Descriptor p, Descriptor q) 85
16 int compareDistance(Descriptor referenceNode, Descriptor a, Descriptor b) 86

Chapter 1

Introduction

1.1 Motivation

The Internet has given rise to large scale distributed systems that offer a wide range of services to end-users. Many of these services could be modeled in a simple and intuitive way by identifying information producers on one side, information consumers on the other and the system that interconnects the two in-between. Usually the consumers of the information are interested in parts of the whole information. This model is called in general a publish-subscribe model where the producers are called publishers and the consumers are subscribed to parts of the produced information and desire to get notified of new information on their subscriptions. Systems that use such a paradigm are also called by the name of distributed event-based systems [31].

1.1.1 Popularity of Publish-Subscribe

Such communication paradigm has taken a central role in many applications that have now come to define core functionalities of the Internet. For example, **RSS feeds** offer a way for users to subscribe to changes in the information published on certain web pages that they are interested in. The client applications then periodically check into this feed to see if there is new information recently published and if there is, they pull it locally for the end-user to consume it. It offers a nice way for end-users to keep in touch with news channels, scientific articles etc. The underlying model is quite simple to understand. The entities that post information on the websites are the publishers, the entities that are interested in the information are the subscribers and the RSS feed reader applications are the infrastructure that connects the former with the later.

On-line social networks have gained a lot of focus in the late years as the most prominent and widely used form of communication, especially among the young population of Internet users. Such network mimic real-life interactions of people like sharing of personal activities, discussions, pictures and gossiping among users via the relationship of 'friending', 'following' etc. Examples of such systems are abundant: Facebook, Twitter,

Google+, Orkut, MySpace, Hi5 etc. The model in this case is based on the notion of friendship. Usually in such systems, all end-users tend to be both publishers and subscribers at the same time. They are publishers for their own profile, while being subscribed to their friends' profiles and activities. The system takes care of informing the users of the latest changes to the profiles of their friends. Given the popularity of on-line social networks, there has been a lot of research going on regarding them, especially on the topic of offloading the stress on the server since these systems become more popular and both their user-base increases and also the amount of usage.

Internet multi-media distribution is another service that is growing in popularity on the Internet. Systems like YouTube, SoundCloud, Grooveshark, Spotify, Vimeo, Metacafe etc. allow users to create channels, post videos on these channels, subscribe to other channels and get notification about new videos in the channels of interest. Most of these systems have also integrated their services with the different on-line social networks to offer better use cases for users who want to share information with their friends, relatives and other real-life connections. The model in this case has video uploaders with their channels, subscribers to these channels and the system that pushes updates of the channels to these subscribers. The integration of the on-line social networks makes the interaction more complex adding the interaction between friends in the on-line social network, fusing in some way the profile in the social network with the channel in the multi-media distribution system.

Stock traders rely on **financial software** to stay in touch with stock quotes, as much as is anybody else that is involved with stock investments. These systems make possible that the stock quotes published get forwarded to the interested entities, based on some search queries that filter only part of the total information according to their investment goals and towards which they have expressed interest in. In general these systems are quite complex, even more than other systems due to the tight time constrains they have between the time a change happens and the time it is delivered to the interested entities, and also the accuracy and sensitivity of the information. The side effects of a malfunctioning system could be in losses of large amounts of money. Again, it is evident how the system falls into a publish-subscribe model.

Other areas where the publish subscribe paradigm can be implemented include **multi-player on-line games**. In this case the subscribers are the players, the events are the movements and actions of the players, while they subscribe on the events that happen on one level or scenario. This is an ever important area of research for many game companies since massive multi-player on-line games are the current trend in this industry. The number of players and the size of the scenarios and consequently the amount of events that needs processing is ever increasing.

In distributed systems where object need to get notified of changes that happen in other object, we could still apply a publish-subscribe system, such as is done in the CORBA Envet Service. In programming languages that apply new **event driven programming** techniques, different objects need to get notified of changes that happen in other objects towards which

they have subscribed. For example the GUI (graphical user interface) programming frameworks like Microsoft's WPF uses a publish-subscribe pattern to model their system or any other GUI programming framework that uses the Model View View-Model design pattern.

There are many more areas to which the publish-subscribe communication paradigm could be applied and would fit their use-case. The whole idea of classifying systems into this kind of paradigm is that we can try to improce all of them with ideas developed for this paradigm.

1.1.2 P2P Publish-Subscribe

The typical approach to implementing such systems is the centralized architecture, having one central entity that takes care of the storage, filtering and distribution of the event to the subscribers, and at the same time it takes care of remembering all the subscriptions. The increase in popularity of publish-subscribe systems has made so that the number of the users of these systems and the amount of information that flows through them is becoming ever bigger. However, the centralized architecture may not be scalable enough to cope with this situation. Of course we can continue to add more storage and more computing power to this central entity, still it does not solve the problem permanently. Also there have been concerns that centralizing information into one entity, be this an organization, enterprise or the government, might lead to the information being misused, easy to steal, censored or temporarily unavailable because a single point of failure creates premises that something will go wrong and this point will fail.

For example most on-line social networks that have a relationship based on the concept of follower and followee can be modeled via a topic-based publish-subscribed model:

- The followees (users of the on-line social network) can be considered as *topics*
- The followers can be the *subscribers* to these topics.
- The updates a followee submits are to be shown to their followers so these updates constitute *events*.
- The updates a followee submits are to be shown to their followers so these updates constitute events and consequently the notifications that the followers get are the *event notifications*.
- Everybody who mentions a followee creates an event on that followee (topic) and therefore the followers of that followee need to be notified.

In this way, on-line social networks like Twitter, or Facebook or any other on-line social network, could benefit by avoiding scalability problems that they have experienced in the past (Twitter downtimes around 2008), or avoid being censored from malignant governments (protests in Egypt, Libya etc., or the government in China), or avoid the whole service being

monitored and controlled by a central entity (commercial company) that could go bankrupt and therefore stop the service, could use user created content and private information for profit without explicit user consent or give user information to governmental agencies (NSA monitoring scandal for example).

The same could be said about other services that are used for spreading news and information and that could be the target of censoring. It is important for us to think about ways to make systems resilient in this aspect too if we want to contribute to creating freedom in the Internet. Decentralized systems, if designed properly, can be hard to censor and trace.

These and many other reasons have made researchers try to make publish-subscribe systems go fully decentralized. This means that it would have to go in a direction where users of the system communicate directly with each other without a central entity acting as an intermediary, therefore creating a peer-to-peer system. In these systems all the users are required to contribute their resources for the global good of the whole system. Potentially such systems could solve many of the issues stated above. There would no longer be a centralized entity to be censored or tampered with. There would not be any scalability issue, at least if the system is designed efficiently. We would not need to increase the central amount of storage or computing power since, first of all there is none and secondly, with an increase in the number of users the amount of storage and computation would increase since each of them would contribute to the system in a equal and fair way.

Some examples of publish-subscribe systems in the real world are:Scribe [7], TERA [2], Gryphon [44], etc.

1.2 Problem Statement

One of the most characteristic properties of P2P systems is the fact that nodes are free to join and leave the system without any prior notice. This process that we have previously referred to as *churn* can have negative effects on a P2P overlay. When nodes go off-line, they can no longer pass on messages to other nodes. If they happen to be of crucial importance to the dissemination protocol, messages could get lost or at least delayed. Depending on the architecture of the overlay chosen for dissemination, there could be a perceived period of absence of service to a part of a group (subscribers to a topic) or for the entire group. In order to be a viable substitute for centralized publish-subscribe systems, their P2P counterparts should provide the same level of reliability and service availability (or comparable).

1.2.1 Ideal P2P Topic-Based Publish-Subscribe

An ideal P2P topic-based publish-subscribe system should fulfill the following properties, as explained also in [36]:

- 100% delivery ratio when in a failure free run
- High delivery ratio under real churn
- Fast recovery and rebuild of the overlay at the end of a churn period so that 100% delivery ratio is restored
- Low degree of the nodes in the overlay
- Relay free routing (topic based connectivity) which means that only subscribers of that topic are involved in routing the messages of that topic.
- Scalability with respect to number of nodes, number of topics, number of subscribers to a topic, number of subscriptions of nodes.
- Fast delivery, low duplication of messages, fair distribution of load.
- Low overhead of overlay maintenance

These properties, unfortunately, are not achievable all at the same time. Some of them are in contradiction with each other and trade-offs have to be made:

Low node degree and relay free routing: if we want relay free routing, nodes have to have connections to nodes that are subscribed to the same topics as they have. So for each topic, there has to exist a link to a node that is also interested in that topic in order for messages not to have to go through nodes that are not subscribed to it. While there are ways to try to find nodes that have the most number of topics in common in order to lower the node degrees if this choice is made, still the degree of nodes will be higher than in the case when relays are used because one relay can serve many topics, if not all (although this might not be a great design choice). Since low node degree is also intertwined with scalability, then relay free routing might also come into conflict with it.

Resilience towards churn and low duplication of messages: As we explained, if the message is directed to a node and this node goes down, the message will not be forwarded to other nodes interested into it. That is why most solutions try to send the message in multiple directions and to different nodes, so that this does not happen. However, as the dissemination proceeds, there is a high chance of nodes getting the same message again from some other node in the network, therefore getting a duplication of a message. It is obvious that the more we try to make the dissemination resilient towards churn, the more duplicated messages are going to appear on the nodes.

Scalability and fast recovery and rebuilding of the overlay at the end of churn periods, or robustness: depending on the type of overlay chosen to be used by a specific system, there can be a fixed number of connections a node has in total or for each topic. The smaller this number of connections a node has, the less able it is to get to know quickly about bigger parts of the network because its view is limited. The larger the number of connections, the bigger the burden on each node and, since this burden grows with the growth of size of the network, the scalability is also affected.

1.2.2 The effects of churn on different overlays

The magnitude of the symptoms perceived from churn can depend on the different type of overlay chosen for the P2P system. Unstructured systems like Quasar [51] are said to be pretty resilient towards churn and that they recover quickly from it, however there are no tests that back this statement up. On the other hand it is true that such systems like Quasar cannot achieve 100% event notification delivery even in the case when the system is static and no nodes are leaving (not better than 95%), so it is undoubtedly going to be worse than that. For these reasons, there is not much discussion in terms of how well unstructured overlays for topic-based publish-subscribe perform under churn.

Structured overlays for topic-based publish-subscribe systems are on the other side of the spectrum with regards to resilience towards churn. Their preferred dissemination structures, multi-cast trees, are quite fragile. These dissemination trees usually put a lot of responsibility on key nodes, like the root of the tree or the forwarders. If these nodes are compromised in periods of churn, dissemination is hindered. Information about topics and subscribers to these topics, is also kept in such key nodes. If these nodes go down, this information goes with them and it has to be re-established on other nodes, which is a costly operation in terms of messages going back and forth between nodes. Maintaining and repairing such structures also takes a high number of messages since the routing tables of DHTs are quite large.

1.2.3 On-line node behavior

In the recent years there has been a lot of research dedicated to making P2P overlays suitable for topic-based publish-subscribe systems that are robust in churn periods. However, most of this research presumes that, given that the nodes are free to join and leave the system as they please, churn is totally unpredictable and the best that we can do is find ways to recover, maintain connectivity and availability of service in case it happens. They try to do this as quickly as possible and using the least number of overhead messages. However, research, such as the ones in [25, 32, 39], has shown that nodes' availability can be predicted to a good extent, given knowledge of previous behavior. The point made in these works is that nodes exhibit recurring patterns in their connection and disconnection to the system, or what some of them refer to as *regularity*. From intuition, it is also logical to believe that, since nodes are controlled by their human users, their on-line/off-line behavior would follow the same patterns of the users behind them. Users have their daily and weekly patterns they follow.

These patterns arise from the cycle of night and day, the working hours, holidays, weekends etc. Users follow these patterns and are these patterns that "force" them to also have some patterns in their on-line behavior. For different systems, depending on their use, there could be different patterns. For communication or news solutions nodes might recurrently be on-line during working hours or the region they belong to. For file

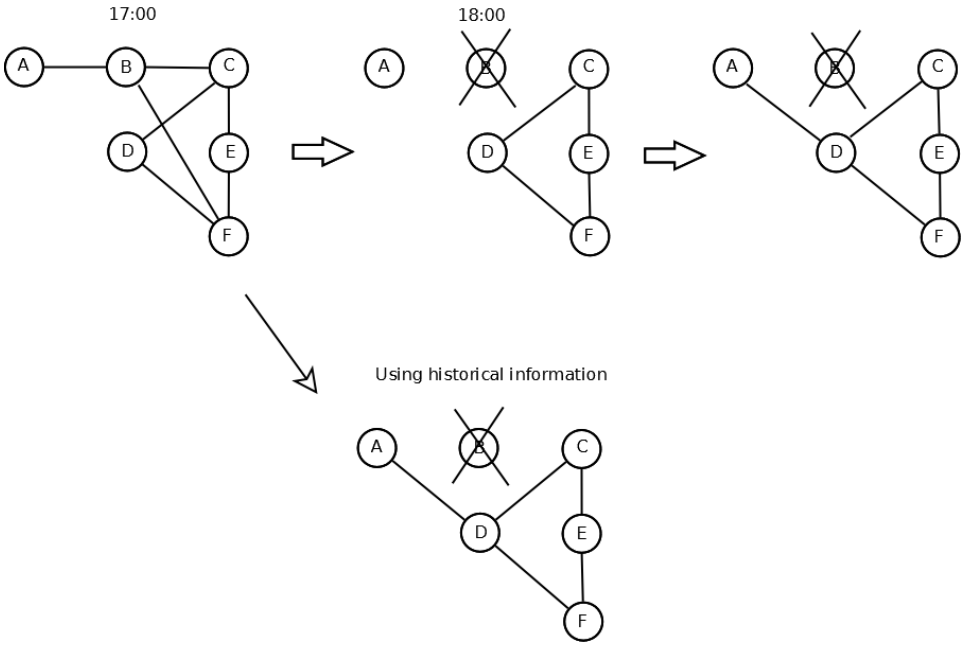


Figure 1.1: A simple overlay demonstrating the disadvantage that arrives if node B, which will go off-line in 30 min, is chosen as a neighbor, versus equivalent node D which will be on-line (most probably) for several hours.

sharing or media streaming solutions, they might be on-line during non-working hours for the region they belong to. On-line social networks might have some different patterns where users take advantage of breaks during the school hours, working hours, non-working hours, weekends etc. One aspect of such patterns in that they are recurring, same as the routine that governs the lives of the users behind them. Every day, week or other repeating period is approximately similar to the one before it. This property of nodes' connectivity can be exploited so that we are better able to select the nodes to which each node connects (neighbors of that node).

All of the overlays suggested by the research done in [7, 8, 17, 20, 26, 34, 36, 46] and many other works do nothing in order to prepare ahead of time for periods of churn, and therefore their only option is to react once churn has happened and the damage to the connectivity of the overlay is already done. The selection of neighbors from nodes that have a high probability of not being on-line in the near future could lead us to having an overlay that could experience disconnection shortly. Same problem is present when selecting the neighbors of the nodes in order to repair a broken overlay. During such periods the overlay might form clusters and messages or events might not be routed to the desired recipients, therefore undermining the service availability.

While it is true that churn in P2P systems is inevitable, it would be a good idea to avoid it or prepare for it, if this is possible. An example is given in Figure 1.1 where we have a small overlay. Let us consider the case when each of the nodes is at the same time a topic to which other nodes can subscribe (sort of like the case of a Twitter-like system). From

previous statistics we know that there is a high probability that node B will go off-line at 18.00, while node D has a high probability for example of being available until 22.00. Let's also consider that A has had the chance of connecting to each of the equally important nodes B and D and has selected B. If node A were to disseminate any message at 18.00 which would be of interest to nodes C,D,E or F, they could not possibly receive the message. In the same way, node A would not get any messages from the other nodes and therefore it perceives a loss of service. In the case we had chosen D as a neighbor of A this would not have happened. If we take advantage of the historical information we can gather about the nodes, we can take steps into assuring that the overlay will have a high chance of achieving continuous connectivity and therefore attaining uninterrupted service.

Inspirational Work

As a starting point for this thesis served the results achieved in [32]. The work presented in that article is concerned about replica placement in DHTs that are used for distributed storage of data. The problem discussed is that under churn in DHTs, replicas have to be relocated to other nodes in order to maintain a certain replication factor for resilience and this creates traffic since the whole data has to be copied over the network.

Many previous works have used patterns of nodes' availability to create strategies of putting replicas on nodes that are most probable to be on-line in complementary moments in time, that have adjacent ID numbers, most available etc. The paper [32] proposes an alternative approach that focuses on nodes regularity in their connection to the network for creating replicas.

But first of all, what is **regularity**? As defined in [32]: If we divide time into periodic intervals (day, week etc.) and these intervals into timeslots of a certain amount of minutes, then a node is regular in one timeslot if the historical ratio between the node being on-line and off-line in this timeslot, is more than a certain threshold γ. So in general terms it means that if a node is regular in a specific interval, there is a high probability(proportional to γ) that it is going to be on-line during that period.

The strategy discussed in [32] tries to take advantage of this property of the nodes in order to minimize the traffic used for replica relocation in periods of churn. This requires that nodes keep a set of candidate nodes for replicas for each object. At least k nodes of the candidate set should be on-line at any given point in time (with high probability, there are no guarantees that they will be). If such k nodes exist, they become the replica set for the object, otherwise temporary nodes are used to fill the missing places. To backup the motivation behind this approach, the authors show also tests on regularity patterns of nodes in eDonkey and Skype. It is shown that a considerable amount of nodes exhibit regular behavior during various time intervals in a day or week. We will make our own analysis of these traces later in the thesis in order to draw our own conclusions and have a better understanding of this property of the nodes. We try to identify the various parameters that affect the perception of nodes as regular according to this method.

This new strategy of replication is tested and the results shown in [32] are promising. By using this technique, the overall bandwidth consumption during a test run is lowered about 20 times in total and up to 6 times per node. This means that the regularity property of the nodes is a reliable one and that we could take advantage of it to improve our overlays.

1.3 Goals

Being able to take advantage of the historical information about nodes' up and down times is not a trivial issue. This is especially true for the case of topic-based publish-subscribe systems where no research has been looking into this. To begin with, we have to consider what would be necessary in order to achieve such a thing:

- First, we need to investigate if nodes exhibit consistent patterns in their availability and quantify this, together with the parameters that influence perception of such patterns.
- Next we need to find ways to collect information about other nodes in the system and create a list of nodes that would be most interesting to the node's own pattern.
- Finally, we need to put this information to use by trying to improve an already existing protocol and analyze the results to see the benefits that could be gained by using this approach.

Improving an existing overlay by using this property of the nodes requires us to fix some goals in this direction also:

- The overhead in terms of number of messages and bandwidth consumption for exchanging regularity information should be kept as low as possible.
- The overhead of computing the list of nodes that a node is going to select as its neighbors should not be high since this computation has to happen at every cycle the nodes communicate with each other and when they try to improve the overlay.
- The message delivery should be improved, but not affected in a negative way.

All of these goals come with some trade-off since what we are doing is adding extra communication and computation to an already existing protocol. The most important trade-off that should be discussed is the one about load balancing and fairness. Every time a property of a subset of the entire set of nodes in a system is exploited, there is a slightly higher burden on those nodes. For example in file sharing systems, nodes that have a higher bandwidth available are used more for transfers than nodes with lower bandwidth. The same thing could be said of the inspiration

work we presented above [32] where regular nodes have a higher burden of storing more information compared to nodes that do not posses this property. Hence the trade-off between the benefit that the whole system gets from such an exploitation of a subset of the nodes and load balancing.

What we will not be dealing with in this thesis is any issues regarding privacy or security attacks that could be enacted towards our solutions. These issues are outside the scope of our work and therefore are to be dealt with by other research works.

1.4 Contributions

Since the field of taking advantage of statistical information about the nodes in P2P systems is relatively new and under-explored, this thesis is firstly focused on identifying the regularity patterns that the nodes exhibit in popularly used systems like Skype and KAD. For this reason we have analyzed traces of these systems in order to determine what percentage of the nodes are exhibiting the property we are interested in exploiting and how different parameters influence this number. We also have done this analysis in order to determine what would be the optimal parameters to use in other systems that might try to take advantage of the regularity property.

As a second contribution of this thesis we have created a generic service for use in P2P systems, called GRID. It is targeted towards identifying the regularity pattern of the nodes themselves and discovering other useful, regular nodes in the system. This service is configurable enough that it can be used by different P2P applications which are able to take advantage of the regularity property of the nodes. As part of this contribution, we also offer an idea of how the regularity information can be maintained and exchanged in an efficient way among the nodes in the system.

Thirdly we have created a proof-of-concept implementation of an application that consumes GRID and which tries to improve message dissemination in a P2P topic-based publish-subscribe system such as PolderCast [36] is. The contribution in this part is multiple:

- We configure GRID for the specific needs of an application aimed towards aiding dissemination in PolderCast using regularity. At the same time this shows the way that could be used to configure GRID to other applications also.
- We see if there is any benefit in using regularity information to improve the dissemination in this specific case. If successful, this could serve as a motivation for trying to take advantage of regularity to improve other systems in the future.

1.5 Assumptions

In trying to achieve the goals we posed to ourselves in Section 1.3 with our contributions, we had to make the following assumptions:

1. We assume that **the underlying network is reliable**, meaning that once a message is on the fly, it is assured to reach its destination if it is on-line. Our contributions take advantage of existing works which require this assumption in order to operate. Also, this allows us to conduct simulations without taking care of introducing link failures.

2. Distributed networks lack the property of globally synchronized clocks and clock drift is a real issue. Our assumption is that this drift is not longer than the length of a timeslot (the concept of timeslot is briefly described in Section 1.2.3).

3. We also assume that nodes are able to keep track of the times when they are on-line and when they go off-line, even in ungraceful exits.

We believe such assumptions are reasonable to make.

1.6 Document Organization

This thesis is organized in eight chapters.

Chapter 1: - which constitutes the introduction, tries to establish the motivation for this thesis, explain the problem we are trying to solve and briefly present the contributions of this thesis

Chapter 2: - here we explain the concepts and the background information needed to understand the work of this thesis and the field of research to which it belongs.

Chapter 3: - goes through some of the most recent and prominent pieces of research in the field of P2P overlays for topic-based publish-subscribe systems and also works related to churn and regularity.

These chapters lay the foundations for the following ones, which constitute also the main part of the work of this thesis:

Chapter 4: - this chapter of the thesis is about the a**nalysis of the availability traces** of Skype and KAD and their respective nodes' regularity patterns. We try to establish the main variables that influence the number of regular nodes at any moment in time. Also, we try to determine which are the parameters that a system which tries to take advantage of regularity information can tweak to achieve different results.

Chapter 5: - is focused on the introduction of a new generic service for P2P systems called **GRID** aimed towards regularity.

Chapter 6: - describes a **proof-of-concept implementation** of trying to improve message dissemination in the case of PolderCast and taking advantage of GRID. In this part we will also explain what the different trade-offs of such an implementation are. Experiments are conducted to verify our hypotheses.

Chapter 7: - is a summary of what was achieved in this thesis. We also draw some conclusions based on the results of the experiments conducted.

Chapter 8: - Future work.

Chapter 2

Background

Different from other communication models where interaction is direct via a request-reply concept, publish-subscribe is more decoupled. Structured events are published by the entities called *publishers*. On the other hand, the consumers, *subscribers*, express their interest in a subset of events via *subscriptions* [11]. As an example, in a stock event notification system, a subscription could be to all the quote changes of IBM, or to all the quotes that have risen by 5% recently etc. It is the responsibility of the system itself to make possible the matching of the *subscriptions* to the *events* that are published by the *publishers* and make sure that all the *event notifications* are delivered to the intended recipients. The main properties that can be achieved by such systems are described in [11, 13]:

- *Time decoupling* - The entities communicating with each-other, are not required to participate in the interaction both at the same time. This translates to the fact that publishers might publish events at different times, even when the subscribers are not available. Also subscribers might receive event notifications even for events that have been published while they were not available.
- *Space decoupling* - Publishers and subscribers do not need to know each other prior to their interaction. Neither of these entities needs to keep track of the connections to each other or know how many of each is there in the system.
- *Synchronization decoupling* - Event production and delivery happens in an asynchronous way. This means that the publishing of events for publishers and the receiving of event notifications for subscribers are not blocking operations.

2.1 Categories of Publish-Subscribe

There are several types of publish-subscribe systems. Here are the main categories as described in [11], in an increasing order of complexity:

- *Channel-based:* In this category, the channels are defined by their respective names. Publishers publish events to these channels and

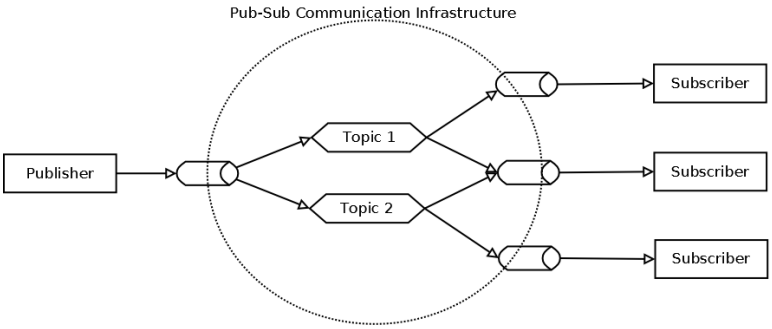


Figure 2.1: Visual conceptual representation of a topic-based publish-subscribe system

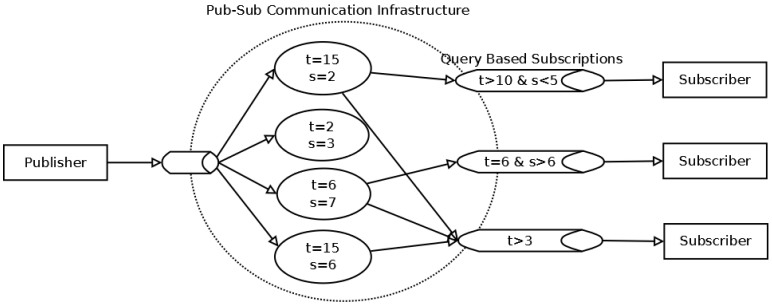


Figure 2.2: Visual conceptual representation of a content-based publish-subscribe system

subscribers subscribe to them with the desire to receive all events that are being published in a channel. This rather simplistic scheme is used in the CORBA Event Service [14].

- *Topic-based:* In these kind of systems, events are characterized by exact properties, each of which constitutes a topic to which subscribers can show their interest into (subscribe). These topics are discrete, meaning that the information can be easily determined if it is part of one topic or another. This means that the problem of matching subscriptions to publications is relatively easy (Figure 2.1). Many systems can be modeled in such a way: Twitter, RSS feeds etc. This will be the type of system that we deal with in the course of this thesis.

- *Content-based:* Here the information has some attributes with a value for each attribute. A subscription would then be a query that filters the values of these attributes to events with attribute values in which the subscriber is interested. Therefore these types of systems require relatively sophisticated query languages that are capable of sifting through the information in an efficient and precise manner. This means that these systems are more complex in nature than the topic-based ones, but at the same time they offer more complex applications to be built on top of them. A simple visual representation can be seen in Figure 2.4. Financial systems are a very good example of such approach where financial dealers submit specific queries of interest and get notified specifically for the events on these queries.

- *Type-based:* Explicitly designed for object-based approaches, this type of system allows subscribers to express their interest in receiving notifications on certain types of events. The queries can be either on the types of the objects or on their attributes and methods.

There are also many other minor variations of the categories above, however they do not introduce major changes to the concepts above. While what we discuss in this thesis might be applicable to any of the categories presented above, we focus mostly on the applications atop topic based publish subscribe.

2.2 Architectures of Publish-Subscribe Systems

Independent of the category of the publish-subscribe system, the main objective of every single one of them is to efficiently deliver notifications to the subscribers of the appropriate events to which they have subscribed. For achieving this, different types of architectures can be used. To put the work introduced in this thesis into perspective, we briefly introduce various architectures, while focusing mostly on peer-to-peer ones which will also be more relevant to the contribution in this thesis.

2.2.1 Centralized architectures

The simplest way of implementation for publish-subscribe systems is using a centralized architecture. In this case, a single server node could act as a broker for the events (Figure 2.3). The publishers would send their events to the server and the subscribers would send there their subscriptions. The server would then match the events to the appropriate subscriptions and send notifications to the corresponding subscribers. The server implementation is relatively simpler given the fact that the server has knowledge of all events and subscriptions.

This architecture relies on a single central machine, it means that it also has a single point of failure. If the server fails, the entire event notification service goes down. It is also equally important to note that the resources of this central machine are limited, there is only a certain amount of clients

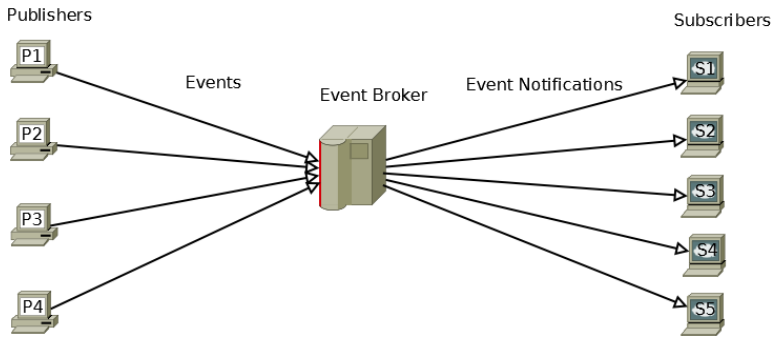


Figure 2.3: Centralized architecture with one server

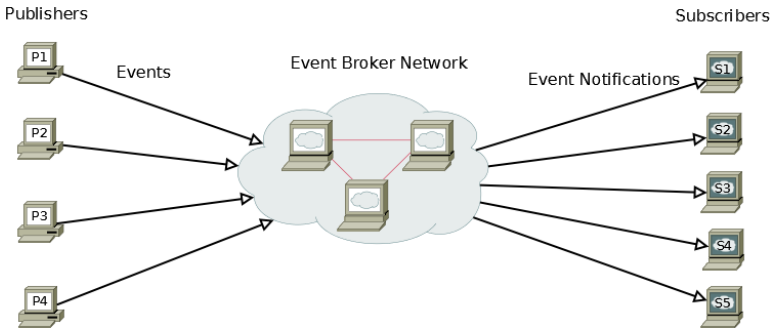


Figure 2.4: Centralized architecture with network of servers

it can handle, therefore it constitutes also a performance bottleneck. Given the scale of the Internet and the scale required for today's services, this kind of architecture is not well suited.

2.2.2 Distributed Event Broker Network

In order to overcome the scalability and performance issues, there have been developed systems that, instead of the centralized single event broker, have a network of servers that function as event brokers. These event brokers communicate with each-other in a peer-to-peer fashion and collaborate together (Figure 2.4). This is the most common architecture used in Internet scale publish-subscribed systems developed today. Such an architecture is used in systems like Gryphon [44] and Siena [6].

The server network can be organized in many types of overlays that are typical in peer-to-peer networks, like ring, hierarchical etc. Peer-to-peer publish-subscribe systems will be explained in a more detailed way in Section 2.2.3.

2.2.3 Peer-to-peer Publish-Subscribe

One type of architecture that is gaining in popularity lately is a full peer-to-peer (P2P) implementation of publish-subscribe systems. This type of implementation tries to do without having a single entity that takes the bigger burden of the load. They try to have an even distribution of load among participants in the system.

Each of the participants in the system is called a node. Nodes form connections to each other and this way they form some application layer networks that are called *overlays*. As the name suggests, overlays lie on top of the physical network that makes it possible connecting to the nodes. In a P2P system, nodes are usually all running the same piece of code, and therefore they share equal responsibility for the operation of the whole system. In the case of publish-subscribe that translates to the fact that there is no distinction between the different actors: subscribers, publishers and event brokers.

Going from a centralized architecture to a peer-to-peer (P2P) one solves the issue of scalability in a well designed system, however it increases the

overall complexity of developing, managing and controlling such a system. When there is a central entity which is responsible for all the dissemination of the messages, we can be sure to deliver the appropriate events exactly and only to the clients that are interested in them and in an easy manner because we are able to know exactly which clients are subscribed to what and which clients are on-line or not. When we take away the centralized entity, we lose this luxury of having a "birds' eye" view of the whole system. Once we go into the P2P realm, knowing about every entity present in the network and the specific interests of each of them is virtually impossible. Of course we could get to know about every node in the system and keep this information at each of the nodes, however this type of system would not be very scalable. Nodes in P2P systems have to rely only on local information in order to create connections to other nodes and it is only through messages exchanged with these nodes that they are going to get to know more about other parts of the network.

Another aspect of P2P networks, especially those that are not managed by any centralized entity (fully decentralized), is that the nodes taking part in them might come and go from the system as they please, without having to obey any predefined schedule, or even without notifying about their leave. This process of nodes coming and going from the system is commonly referred to as 'churn'. Churn can have great impact on disrupting the functioning of a P2P system and therefore has remained in the focus of researchers that are trying to build such systems. We will go into more details about the effects of churn in different implementations and about how different systems try to protect against these effects in Sections 2.2.3, 2.2.3 and 2.2.3.

There are three prominent types of fully decentralized P2P networks based on the way nodes are connected to each other in the overlay:

1. Unstructured

Nodes all have the same role in the system, so they make no distinction towards which other nodes they connect to, as this choice is made at random. This randomness gives these networks great robustness against churn because any peer is as good as any other since there is no distinction in the way neighbors are chosen.

The lack of structure on the other hand makes it hard to find the nodes or information required. The main form of communication used in this type of systems is via *flooding* (a form of epidemic dissemination). The messages are flooded into the overlay, often having attached a number called time-to-live (TTL) which specifies how many nodes are allowed to forward this message further. Each of the nodes emits a *fanout* number of messages to pass to its neighbors and decrements the TTL by one. The node that gets the message with a TTL=0 does not forward the message anymore. The bigger the TTL, the more nodes are likely to get the message, but the more overall bandwidth is consumed and the higher is the number of nodes receiving the message more than once.

Another form of dissemination, still epidemic in nature, is random-

walks. This is equivalent to flooding with a fanout=1. This method is most often used to spread search queries when looking for information. Usually, multiple random walks are started in parallel by the node that starts the dissemination in order to improve the chances that the message will be delivered to its intended recipients. If the message contains a search query, parallel random walks help the search be more complete.

These forms of dissemination do not guarantee that the node to which the message was intended for will receive the message. In the case of a publish-subscribe system it would translate into a situation where not all the nodes that are subscribed to a particular event, receive notifications about that event. This is why these forms of dissemination are also called probabilistic forms of dissemination. On the other hand, there is also a probability that nodes that are not interested in some events, might receive notifications about them anyway, therefore creating unnecessary traffic on their side.

Several attempts have been made to construct publish-subscribe systems based on such overlays. Some examples could be: Quasar [51], the system proposed in [9] or [10] etc. However, even in static tests (in absence of churn), these systems fail to have 100% delivery of event notifications to subscribers, let alone under churn. For this fact, unstructured overlays are not the preferred choice for designing P2P publish-subscribe systems.

2. Structured

In this category of systems, the nodes are designed to try and find a specific place in the overlay, creating a desired logical topology. The presence of a structure helps nodes be able to reach any other node they desire and the queries will always be able to find the resources that are present in the system if they exist, at least in a static environment, without churn. There have been many structures proposed for this type of P2P networks, however the most prominent one is the distributed hash table (DHT).

A DHT is similar in functioning to the way a normal hash table functions. The nodes are assigned an ID that is unique in the whole network. Each of them keeps a table of indexes that keep one node per each prefix. The information stored in such a table is dense about the closest neighbors to a node and becomes more and more sparse about the nodes that have higher ID distance. This is usually called a 'finger table'. At each prefix length, it keeps the node that it finds with ID that starts with that prefix and has the lowest value. For example, if an ID is represented as a 3 character string of numbers in base 10 and the node's ID is 431, then it has 1 link to each of the prefixes 430-439, 1 link for each 0XX-9XX. So in total it would have a link to 18 other nodes in the system. In this way, if a message is needed to be routed to a node with ID 219, the node 431 would look inside its finger table and send the message to node starting with 2 it has closest, let's say 200. This node would pass it on to 210 and then this one would have the destination in its routing table and therefore it would deliver the message directly to it. So it took 3 hops to get the message delivered. The number of hops needed to deliver a message in our

example would be $log_{10}(N)$ (since we are using decimal strings), where N is the maximum number of nodes in the system. The maximum number of hops needed to deliver a message however depends on the implementation of the DHT, and more specifically on the size of the 'finger table'.

The bigger the base we use for constructing the ID, the bigger the routing table has to be, but the lower the number of hops needed, so there must be a trade-off between these parameters. Usually the IDs are constructed using hexadecimal strings. They are created by consistent hashing functions like SHA1 or others and they are 128 bit strings or more, which translates to 32 hexadecimal characters. This means that the node will have to maintain 32 rows in its routing table (like in Pastry) and for each of these rows, there will be 15 nodes. This would translate to 480 total links to other nodes in order to have efficient routing. This might not be a problem in itself, unless considering that for systems like topic-based publish-subscribe built on top of DHTs, there might be the need to maintain such a table for each topic the node is subscribed to. The mere number of connections that are required to be maintained in order to have efficient routing of messages and the number of messages that have to be forwarded to other nodes, would make DHTs not suitable for such systems. It would especially be a problem maintaining such huge routing tables in the face of churn as they become very fragile. This makes these systems more suitable for supervised P2P systems, such as ones composed by machines that are maintained by a company or some other entity.

There have been many attempts at building publish-subscribe systems on top of overlays that use DHT. Some examples of such approaches are Scribe [7] and Bayeux [52]. We talk more about such systems in Section 3.1.

3. Hybrid Systems

There exists a third type of P2P overlays that try to take the best of both worlds. These systems try to take advantage of the robustness of unstructured P2P overlays and introduce some form of structured layers on top of them in order to achieve reliable message delivery. Without churn, such systems are capable of achieving 100% delivery of messages to the intended recipients.

Usually these systems are tailored to specific applications. For example some are specifically designed to work for topic-based publish-subscribe systems, others are designed to work for content-based publish subscribe systems or even for distributed data storage (Freenet for example).

These types of systems are proving to be very adaptable in the face of churn and quite appropriate to be used for Internet-scale applications. One of the most useful aspects of the designs of such systems, is that usually they are based on modular designs. The usual approach is to have multiple layers of protocols creating their own overlays, with unstructured ones at the base providing high churn resilience and structured ones at the top, feeding from the connections of the layers beneath them, that provide the overlay needed for reliable dissemination.

One of the goals of this thesis (described in Section 1.3) is to improve

the dissemination of an existing protocol for P2P topic-based publish-subscribe. For this reason we have chosen to follow a modular approach similar to the one in a hybrid protocol called PolderCast [36] and we explain the protocols we extend in Section 2.4. The extensions to these protocols are explained in our proof-of-concept implementation in Chapter 6.

2.3 What is regularity?

We briefly explained what we will define as regularity earlier in this chapter. Now we further elaborate on this concept. The concept of regularity we are using in this thesis is a combination of the regularity concept used in [32] and the concept of high availability in [40] and [38]:

cycle of recurring behavior (or shortly *CRB*). We divide the entire line of time into intervals that repeat themselves, which we call CRBs. These intervals have to have a meaning in real life in order for them to be a useful way of dividing time for our purposes.

The period comprised in a CRB is called the **CRB duration** or **length of a CRB**. The CRB duration should correspond to periods in human life that are considered to be recurring. For example a CRB could be of the length of a day, a week, a month, a year etc. However, the two periods that exhibit real recurring behavior in real life are a day and a week because of patterns of day/night and patters of working days, weekends etc. Months and years on the other hand have less of a cyclical nature since each month is different from the last in terms of length and what it represents and also, both these periods, are too long for practical consideration. Therefore, in this thesis we will deal with CRBs with the length of a day and a week. As a consequence we will be talking about *daily regularity* and *weekly regularity* respectively.

We refer to a single instance of a CRB by the notation **CRB window**. The CRB is an abstract concept and when we refer to it, we do not refer to a specific CRB window. Instead we refer to the recurring period the CRB represents.

To better understand what a CRB is, let us take an analogy between the concept of CRB and the concept of a week in everyday life. A week is a recurring period of time that starts every Monday and ends every Sunday, lasting exactly 7 days. By the end of Sunday, the week restarts. The whole of time is divided into weeks. So the concept of the CRB, in this sense, is similar to the concept of a week. The whole of time is divided into CRBs. The CRB has a

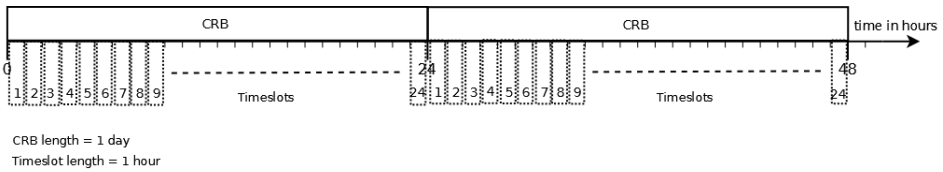


Figure 2.5: Example of how the concept of CRB and timeslot fits into the flow of time

specific duration and the CRB restarts every time the CRB duration has elapsed. In this same analogy, referring to a specific CRB window is the same as referring to a specific week (e.g. week 35).

timeslot We divide the CRB into *units of equal length*, called timeslots. Each timeslot represents a specific amount of time that defines the smallest unit in which we are interested when dividing time. This will be the smallest unit of analysis in regards to statistical information about the nodes on-line and off-line time. It should be noted that length of **CRB** > length of **timeslot**. A node will be considered to be regular or not independently for each timeslot. For each timeslot we are interested in the percentage of time the node was on-line and off-line in it.

If we take as an example the duration of the CRB to be 1 day, and the **timeslot duration** to be 1 hour, then we have 24 timeslots in the CRB. When we refer to timeslot number one, we are referring to the abstract entity that represents the first hour of the CRB. Since the CRB is abstract itself, the first timeslot corresponds to the first hour of every CRB window and not to that of any specific CRB window. Also when denoting timeslot t, t in this case represents the **timeslot number** (in other words, the index of the timeslot if we consider the CRB as an array of timeslots; see Figure 2.5).

regularity threshold A node is considered **regular in a certain *timeslot* t with threshold** γ if the ratio between the time the node has been on-line and the time the node has been off-line in t is greater than or equal to γ. The status of being regular in a certain timeslot is either true or false.

In order to better understand these notations, we have created a representation of the flow of time and how the concept of CRB and timeslot fit into it in Figure 2.5.

The shorter the timeslot length, the more exact information we are able to gather and the amount of information is bigger. This could signify a trade-off because of memory consumption, computation etc. This is one reason why it is not a good idea to choose a very small timeslot length. The other reason is that we do not always want or need the most exact information because then we do not take into account that in the real

world networks there are delays, lost packets etc. which could influence our perception of a node being regular in it or not. The more statistical information available to us, the better the results we can extract are. On the other hand the length of a timeslot should not be large enough to become too attenuated and loose information about specific events. For example a timeslot longer than 1 hour (let's presume $\gamma = 75\%$) could result in a situation where the node could be mostly on-line for the second hour, but for the first hour it is mostly on-line for only 50% of the time.

As with regards to the CRB, the longer it is, the more timeslots it contains and therefore the more memory is required to process it. This is important when deciding the details of the implementation of any application making use of regularity. Also, the longer the length of the CRB, the more time is required to pass until the time of a specific timeslot comes again in the next recurrence of the CRB. This means that the status of a node being regular in a specific timeslot or not, will take a longer time to change. This effect is positive because it makes it unnecessary to update the information about a node's regularity very frequently.

2.4 Gossiping and dissemination protocols used in this thesis

In order to see if the regularity information could provide benefits in terms of improving event notification dissemination, we wanted to create a proof of concept implementation. As such we needed a protocol to improve that is extensible and we needed a way to improve it in such a way that we did not hinder the normal functioning of the protocol since our main focus is to measure only the impact of our implementation in the improving of the overall dissemination efficiency. One such protocol is PolderCast [36]. Therefore we focus a bit more in detail at how this protocol works and the different components that are part of it. This is done due to the fact that later in Chapter 6 we extend some of these protocols and add these extensions to the protocol stack of PolderCast.

PolderCast is a P2P architecture for topic-based publish-subscribe. It is a protocol that belongs to the hybrid overlays that mix unstructured overlays and structured overlays in order to achieve deterministic dissemination without the presence of churn and improve dissemination under churn. To do this, PolderCast uses a stack of three underlying protocols, each of which generates its own overlay, but at the same time the connections in each of the overlays are used by the layers on top of it to improve their own topology.

The structure of PolderCast and its three layered approach is presented in Figure 2.6. The bottom layer is a peer sampling protocol that tries to always have a random sample of the whole network. As a choice for this peer sampling protocol, Cyclon [47] is used. On top of this layer, an interest clustering algorithm is used, Vicinity [46], and it has the duty to find neighbors for each of the topics that the node is subscribed to while trying to minimize the total number of neighbors used to do this. For

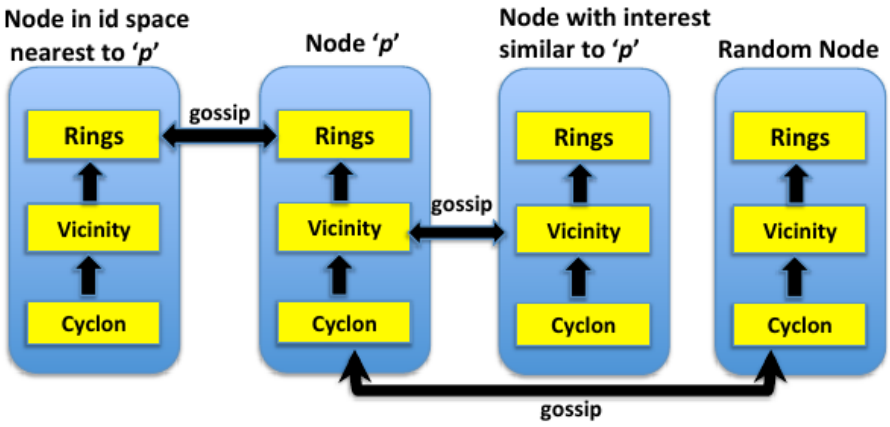


Figure 2.6: Structure of PolderCast's three layered approach

this it tries to use neighbors that have as many topics in common with the current node as possible and that are most beneficial to the layer that is on top of Vicinity. The top layer is the Rings layer, an evolution of the RingCast [49] protocol adapted for topic-based publish-subscribe, which tries to construct a ring of all the nodes in the network arranging them according to their unique ID, introducing in this way some structure. This layer is the one that is the basis for the dissemination of messages.

2.4.1 Cyclon

Cyclon [47] is presented as an enhanced shuffling protocol for overlay maintenance and as a peer sampling service. It is an evolution over basic shuffling algorithms like the one in [41], a relatively simple epidemic algorithm. In such kind of algorithms, a node knows a small set of c other nodes in the system, called a *view* of the system, and every time a fixed amount of time passes, it communicates with one of them and tries to exchange part of its set of neighbors with it. So the set of neighbors a node has knowledge of is continuously changing. The operation of exchanging neighbors is considered a *shuffling*.

Cyclon extends basic shuffling by assigning an age to the descriptors of the nodes in the system. This age in a descriptor d represents the number of gossiping cycles since any node communicated with the node represented by d. When a node is going to initiate gossiping with a node in its view, it does so with the one that has the oldest descriptor. The age of the all descriptors is increased by one every gossiping cycle, while it is set to 0 when it is gossiped with the corresponding node it represents. This ensures that all nodes in the view are cycled through in a periodic fashion. At the same time this process has the effect of enhancing garbage collection since by gossiping periodically with all nodes in the view, it is sure that descriptors of dead nodes will be found out more efficiently and therefore can be removed.

Since we are not going to modify any of the inner workings of Cyclon but instead we will use it as a black box, we are not presenting its algorithm

here as it is identical to the one presented in [47]. This protocol is tested and guaranteed to offer a connected overlay under no churn and it is proven that this algorithm cannot divide the overlay into disconnected clusters. It is also proven that it produces a small world network, meaning that the average path length between any two nodes in the system is around 5. With view length of 20 and shuffling lengths of 8-18 the converging speed (convergence is the point where no significant changes happen in the network) is between 30-40 cycles of gossiping.

The view length is important for robustness in conditions under churn. The larger the view length, the better: less clusters are formed during periods of churn and recovering from churn events is faster. Of course there is a trade-off between the number of connections that we want to put on a single node (view size) and the robustness that we want to achieve. For example, some tests in the paper where Cyclon is presented show that a view size of 100 is very robust with no clustering of nodes even when the number of nodes removed from the system approaches 100%. However, a view size of 100 might not be very desirable for some applications. For example in PolderCast, where the view sizes of all its inner protocols affect the scalability of the entire protocol, the view size chosen was 20 with a gossip length of 10. However, these parameters can be tweaked if needed.

2.4.2 Vicinity

The second layer of PolderCast, Vicinity, is presented as part of the PhD thesis of S. Voulgaris [46]. This protocol is distinguished from peer sampling protocols in the fact that, while peer sampling protocols like Cyclon try to get a random sample of the network, having no preference towards which nodes are considered as neighbors, Vicinity tries to organize nodes into a specific topology by making each node try to look for specific properties when selecting neighbors. However, when forcing clustering of nodes in the network, one loses robustness in periods of churn because nodes are less aware of what happens in other parts of the network outside the cluster where they belong. That is why Vicinity is designed to be used in tandem with Cyclon, which provides connectivity, robustness and fast-recovery when facing churn. This is done by having Vicinity use also nodes from the view of Cyclon when trying to repair its own view.

The Vicinity protocol is also a gossiping protocol where nodes exchange information with each other about their neighbors and their own interests (application-specific information). In this aspect, the exchanged information is almost identical to Cyclon, with the addition of the application-specific part. This could be the list of topics the node is interested in, the channels the node is subscribed to etc. Nodes maintain a view of length $l_{Vicinity}$ of their neighbors in the Vicinity layer and gossip only a subset of a fixed length $g_{Vicinity}$ (gossip length) of this view.

At the basis of the protocol is a selection function $S(k, p, D)$ (where p is the node to be used as reference, D is a set of nodes and the output is k nodes from D that are most suitable as neighbors of p) that gives us a way of sorting nodes in a specific order when having a node in mind as reference,

and then keeping the k topmost. This is called a peer proximity metric because it sorts nodes in order of how "close" their application-specific informations are. It is mentioned in [46] that Vicinity is most efficient when the selection function exhibits the property of transitivity. This means that if node p is a good choice for node q, and node s is a good choice for node p, then S should also be a good choice for q. The transitivity property can be used during gossiping to improve the views of nodes.

In a similar fashion to Cyclon, nodes periodically gossip with their neighbors every-time a specific amount of time has elapsed. The protocol for Vicinity, as presented in [46] is shown in Algorithm 1.

Algorithm 1: Algorithm of the Vicinity protocol as presented in [46]

1 Increase the age of each neighbor by one;
2 Select neighbor q with the highest age among all neighbors, and remove it from the *Vicinity* view: $V_{Vic} = V_{Vic} - q$;
3 Merge the *Vicinity* and *Cyclon* views in one: $V_q = V_{Vicinity} \cup V_{Cyclon}$;
4 Add own descriptor with own profile and age 0 to the merged view: $V_p = V_p \cup p$;
5 Strip down V_p to its $g_{Vicinity}$ best neighbors for q, by applying the selection function from q's perspective: $V_p = S(g_{Vicinity}, q, V_p)$;
6 Send V_p to peer q;
7 Similarly, receive V_q from peer q, containing a set of (up tp) $g_{Vicinity}$ descriptors known by q, optimally selected for p (by calling $S(g_{Vicinity}, p, V_q)$);
8 Merge the *Vicinity*, *Cyclon* and received views in one: $V = V_{Vicinity} \cup V_{Cyclon} \cup V_q$;
9 Rebuild the *Vicinity* view by selecting the best $l_{Vicinity}$ neighbors from V: $V_{Vicinity} = S(l_{Vicinity}, p, V)$;
`/* Node q on the other hand executes parts from step 3 and onwards upon receiving a request to gossip from p. */`

Again, the age part is for the same reasons it was introduced in Cyclon. Consistently cycling through the nodes in the view provides always with a fresh copy of the descriptors of each node. Some of the nodes might have a better view to share since the last time they were gossiped with. At the same time this mechanism makes sure that dead links are removed from the system.

It is proven by the author that the protocol is most efficient when used in combination with a peer sampling protocol, such as Cyclon, for when used alone it is not able to always achieve the desired topology due to lack of information from all over the network. It is shown that for some sample proximity functions, the convergence is achieved in around 20 cycles. Of course, view size and gossip length matters and the bigger these values are, the faster the convergence is achieved, however the higher the burden on the nodes. There are no extensive tests to find the optimal values for these parameters, however values around 10-20 are suggested for both $l_{Vicinity}$ and $g_{Vicinity}$.

In PolderCast, the proximity function is tailored in order to find nodes in the network that have similar topic subscriptions to node p, trying to achieve k (a configurable parameter of PolderCast) ring neighbors for each topic. The proximity function is designed to emphasize special interest especially on those topics for which there are no neighbors in the Rings layer, or which are under-covered (have below k neighbors). This process is attempted to be done with only $l_{Vicinity}$ nodes in the view, trying to get nodes that have multiple common topics of interest so that the desired k coverage of topics in the Rings layer is fulfilled. The proximity function $S(l_{Vicinity}, p, D)$ selects $l_{Vicinity}$ top nodes after the sorting according to the score gathered by each node calculated by the formula presented in Formula 2.1 (the score a node q would have with the reference point of node p):

$$score(p,q) = \sum_{t \in T_p \cap T_q} |T_p|^{k-|S_t|} \tag{2.1}$$

where T_p and T_q are the sets of topics node p and node q are subscribed to respectively; S_t is the set of neighbors in V_{Rings} (view of Rings layer, Section 2.4.3) that are interested in topic t; k is the desired coverage factor in the Rings layer;

The formula makes sure that the more under-covered a topic is in the Rings layer, the higher the score a node subscribed to this topic will get. The overall score that quantifies the utility node q brings to a node p is a sum of T_p raised to the power $k - |S_t|$, for all topics t both p and q are subscribed to. So each term of the sum contributes a score exponential to the difference between the coverage factor parameter k and the actual coverage of that topic. This gives higher score to the terms corresponding to the topics that have the lowest coverage in the Rings layer. In case node q covers a highly under-covered topic, it would get a higher score from this formula therefore increasing its chances to be chosen as a neighbor in the Vicinity view.

2.4.3 Rings

The Rings layer has two functions: construct and maintain a ring like topology of the overlay and provide the necessary links for the dissemination protocol. Nodes have a unique ID in the system, which could be constructed for example from the hash of the nodes name, IP or combination of something unique. This means that there is a unique and globally accepted order in which the nodes can be arranged according to their ID. Since we are trying to construct a ring, either sorting order, ascending or descending is fine, as long as all the nodes abide to the same rule.

In execution, the Rings protocol could be considered a variation of the Vicinity protocol where the proximity function is designed to find at least 2 nodes, one with ID higher than its own, and one with ID lower than its own, always trying to find the nodes closest to itself. By keeping the predecessor and successor, nodes can help each other in their quest for finding their

ring neighbors since the successor of a node is the predecessor of another making it possible for nodes to find the ones that are closest in the ID space to them.

This layer was first presented as RingCast in [49] as a multi-cast medium, and then adopted for topic-based publish-subscribe in PolderCast[36]. RingCast is appropriate for one topic dissemination. In PolderCast each node might be subscribed to more than one topic, therefore there needs to be one ring for each topic in order for messages of these topics to be disseminated. Having a separate ring for each topic is also done in order to preserve topic privacy and avoid nodes routing messages for topics they are not subscribed to.

The protocol execution is almost identical to Vicinity, with some exceptions that we will see. The algorithm is presented in Algorithm 2.

Algorithm 2: Algorithm for the Rings protocol, presented in [49] and [36]

1 Increase the age of each neighbor by one, all neighbors in the view for all topics;
2 Select neighbor q with the highest age among all neighbors, and remove it from the *Rings* view: $V_{Rings} = V_{Rings} - q$;
3 Select random topic t to gossip from the set $T_q \cap T_p$;
4 Create a set V_p of neighbors from Vicinity, Cyclon and Ring neighbors, selecting only those that are subscribed to the selected topic t;
5 Add own descriptor with own profile and age 0 to the merged view: $V_p = V_p \cup p$;
6 Strip down V_p to its g_{Rings} best neighbors for q, by applying the selection function from q's perspective: $V_p = S(g_{Rings}, q, V_p)$ that will give the nodes that have the closest id to q, preceding and succeeding q;
7 Send V_p to peer q;
8 Similarly, receive V_q from peer q, containing a set of (up tp) g_{Rings} descriptors known by q, optimally selected for p (by calling $S(g_{Rings}, p, V_q)$);
9 Merge the *Rings*, *Vicinity*, *Cyclon* and received views in one: $V = V_{Rings} \cup V_{Vicinity} \cup V_{Cyclon} \cup V_q$;
`/* k is the desired coverage for each topic node p is subscribed to */`
10 Rebuild the *Rings* view by selecting the best k neighbors from V: $V_{Rings} = S(k, p, V)$ for each of the topics p is subscribed to since we have one ring for topic;

Usually the k and g_{Rings} are small because we just need one successor and one predecessor per topic. However we usually keep 2 or more predecessors and successors in order to have better resilience in case of churn. The target overlay for each topic should be as the one presented in Figure 2.7. The protocol we just described shows how the ring overlay

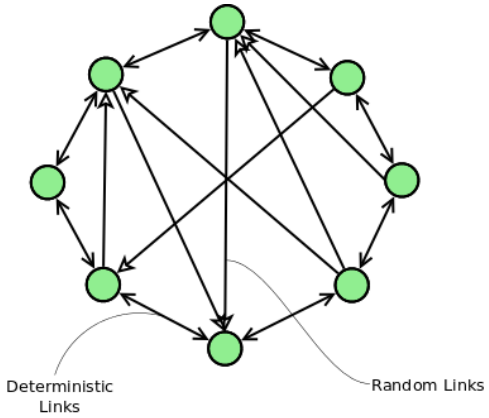


Figure 2.7: Rings overlay for one topic.

is formed, specifically only about the deterministic links that are formed between nodes in order of ID. These links are called deterministic because at any moment in time, by knowing the total set of nodes, there is only one immediate successor and one immediate predecessor to a node ID. However, there are also random links that we present in Figure 2.7. These links are selected from the nodes that are subscribed to the topic of the ring in the union of $V_{Vicinity} \cup V_{Cyclon}$. The importance of the random links is explained in the following section about the dissemination protocol.

2.4.4 Dissemination protocol

The dissemination in PolderCast takes advantage of the deterministic nature of the ring in order to offer full dissemination in conditions where churn is not present. The idea is that, if a node receives a message, it forwards this message to f (fanout factor) other nodes. The node forwards the message to its predecessor and successor, which it gets from the Rings neighbors, and $f - 2$ other random nodes that are subscribed to the topic for which the message is concerned, that the node gets from the union of views of Vicinity and Cyclon. The dissemination protocol also avoids sending the message back to the node from which it received it. Therefore, when the message is received by either the predecessor or successor of the node in the ring, the number of random nodes it forwards the message to is $f - 1$.

A representation of a dissemination scenario for a topic is presented in Figure 2.8. It represents a dissemination in an overlay that has been partially disconnected under churn. This disconnection makes it impossible for the dissemination to reach all the nodes in the ring just by forwarding it to the nodes' successors and predecessors. That is why there was made the choice to forward the messages to random links for the same topic. These random links have a higher chance of making it possible to connect disconnected parts of the ring, therefore allowing the messages to be disseminated also in their region of the ring. The other reason is faster dissemination even in the absence of churn. Messages delivered via

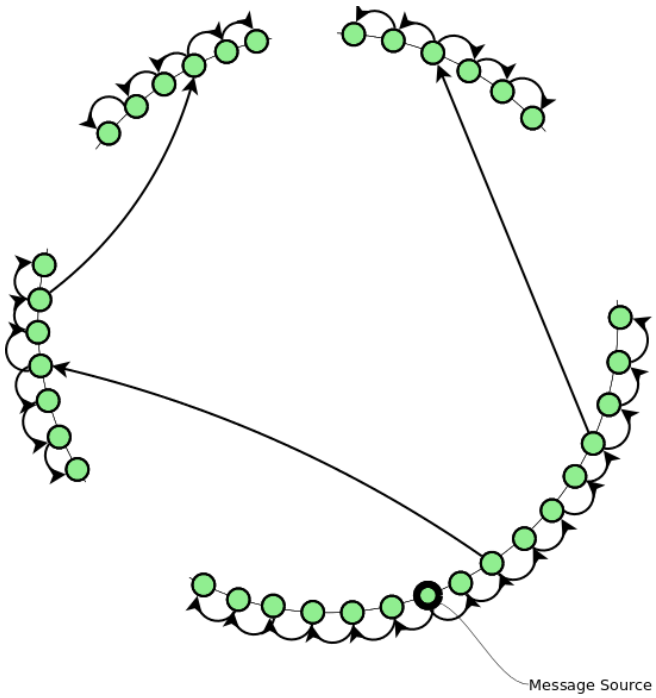


Figure 2.8: Dissemination over the Rings overlay for one topic

random links ignite dissemination in various points of the ring and this is what increases the speed of dissemination. The downside to these random links is that they increase the duplication factor of the protocol as nodes do not know which other nodes have already received the message. This is a trade-off to be made between churn resilience of the dissemination and the duplication factor.

2.5 Evaluation metrics

There are certain metrics that define the performance and efficiency of a dissemination protocol. We present here the main ones that we use in this thesis:

Hit Ratio represents the percentage of nodes that have received the message meant to be disseminated. The maximal hit ratio is 100% when all the nodes that are subscribed to the topic to which the message belongs to, receive it.

Node degree represents the number of connections that a node is actively maintaining. This parameter is important to show how feasible the system would be in real life. Maintaining a large number of connections is expensive for the machines that the nodes represent and therefore can be degrading to performance if it is too big. As an example, torrent applications suggest not putting more than 200 connections as a maximum limit. Node degree is divided into three categories: in-degree, out-degree and overall-degree. Let's take a node p as the node in consideration. The in-degree of p is the number of connections that are not initiated by p, but are

directed to p from other nodes. The out-degree represents the connections which are initiated by p itself towards other nodes. The overall degree is the set of all the nodes connected to p, including those in the in-degree and those in the out-degree. In the calculation of these parameters, all connections in all protocols are considered.

Number of messages sent and received is the number of messages that are received and sent in the dissemination protocol. Also, it represents the amount of bandwidth used for dissemination in relative terms.

Path length of the dissemination represents the number of hops (number of nodes forwarding the message) the message goes through from the moment the it is created at the source of the dissemination, until it reaches an interested node. For a good dissemination system, the path length should be as low as possible to allow for near real-time dissemination.

Message redundancy represents the number of times a message is received by the same node. These duplicate messages are useless as they do not contribute to the dissemination while they consume bandwidth and other resources.

Each of these metrics represents an aspect that is important for a dissemination protocol. The main objective of such protocols is usually hit ratio, however without proper balance of the other metrics, the protocol might be highly inefficient with regards to bandwidth, delay, overhead etc.

Chapter 3

Related Work

For the course of this thesis we focus on P2P topic-based publish-subscribe. It is therefore of importance to discuss existing state of the art works in this field and identify the main directions these works have explored.

3.1 P2P overlays for topic-based publish-subscribe

P2P systems that appear in various articles choose one of two ways to provide message dissemination. The first and earliest one is to not arrange nodes in any specific structure and follow an epidemic type of dissemination. This approach is for example followed by a system like data-aware multicast presented in [1] and Quasar, a probabilistic dissemination architecture using routing vectors installed on nodes to direct messages towards interested nodes [51], among others. Even though the epidemic nature of the dissemination in these systems gives them reliability under churn, however they do not have any deterministic guarantees that the message will be disseminated to all nodes that are interested in it even under a static, churn-less, environment. This fact makes such systems not very suitable for most uses of topic-based publish-subscribe.

Clustering algorithms like SpiderCast [8], StAN [26] or Vicinity [46] try to cluster nodes that have similar interests together while attempting to lower the overall degree of the nodes. The resulting overlay is still an unstructured one, even though the links are not strictly totally random. Having nodes interested in the same topic clustered together, increases the chances that a message destined to this topic, reaches most, if not all, of these nodes. However, clustering can lead to weak links between the clusters which might create fragile links under churn, especially for not so popular topics. Therefore systems like Vicinity or StAN are designed to work in tandem with other protocols that generate random overlays such as Cyclon [47] or SCAMP [15]. Other systems like SpiderCast have tried to balance clustering with random links which are supposed to counteract this bad effect of high clustering. Systems like StAN, Vicinity or SpiderCast fall into the category of overlay creation protocols. These are no disseminating protocols and therefore there is no evidence about how these systems

would fare under churn in terms of hit-ratio.

The second choice that is prominent in various research works is to introduce some form of structure into the dissemination overlay to enable full message delivery under a static environment while attempting to have also good resilience towards churn. Such structures are usually based on DHT and generally revolve around the creation of dissemination trees for each topic. Such structures are constructed in Scribe [7], Bayeux [52], Magnet [17] and Vitis [34] for example. The root of these dissemination trees is responsible for maintaining information about the topic of the tree and of the membership. Also, each tree node p is responsible for disseminating the messages down the tree through the branches rooted at p. This makes dissemination trees on top of DHTs vulnerable to the failure of nodes that are key to the whole dissemination. This makes these structures fragile under churn. When a node p goes off-line, the dissemination to the branches of the dissemination tree originating at p is threatened. In addition, maintaining dissemination trees requires many messages to be exchanged between the nodes, which constitutes overhead.

Except for Vitis, the other works are not tested in the papers where they are introduced with regards to hit-ratio in the face of churn. For Vitis, the authors claim that the hit-ratio reaches and remains 100% even under churn, however such results are debatable because the gossiping frequency used during the simulations is not mentioned. Altering the gossiping frequency can vary the speed with which the churn trace is played. If the churn trace is played slowly enough, then the overlay has sufficient time to repair by gossiping. In [36] a test of Scribe's hit-ratio under churn is done and it is shown that under the specific settings used in that work, this metric achieves quite high values above 96%.

Other approaches implemented in the literature are part of a new type of family. These are semi-structured overlays, or hybrid overlays. In these types of systems, generally there is a mix of unstructured overlays with structured overlays. It sounds logical to try and combine the best properties coming from the two approaches. The unstructured overlay is typically used to achieve better overall connectivity and higher resilience to churn while the structure introduced provides reliable dissemination in static scenarios and improved dissemination under churn. Such approach is used in PolderCast [36] and RingCast [49]. PolderCast is an architecture composed of three layers. One layer is a peer-sampling protocol (Cyclon [47]) which creates a random overlay and provides general connectivity. The second layer is a distributed clustering protocol called Vicinity [46]. The third layer is the one that creates a structured overlay called Rings which is used for the dissemination of messages. The inner workings of PolderCast are described into more detail in Section 2.4. The structure introduced by the Rings makes the dissemination to be deterministic under no churn, which means that messages reach all the nodes that are interested in them. In experiments with churn, with the respective settings, both PolderCast and RingCast perform quite well with hit-ratios on the average around 98-99%.

For this thesis, another concern is in how the protocols behave in

conditions of churn. It is worth noting that none of the aforementioned protocols try to do anything more than react to churn after it has happened. This means that they rely on the assumption that churn is unpredictable. However latest research has been able to identify patterns in on-line node behavior. Such work as the one presented in Section 1.2.3, have taken advantage of statistical information about the nodes *up* and *down* times and used it to improve replica placement in a DHT data store.

3.2 Churn Analysis

Churn, being one of the most important aspects of a P2P system, has been studied in depth in the literature. However, it must be said that studying churn is not easy due to many pitfalls as explained in [45]. The main point is that, in order to fully understand churn in a P2P system, we need to get a global view of the on-line node population in the system and how this population changes over time. This is not an easy task as P2P systems usually do not have any centralized entity keeping track of the nodes that are on-line. Therefore, researchers usually have to deal with limited views of P2P systems that sometimes might not be good representatives of the whole system. The duration of the period in which a P2P system is monitored is also important to understand how churn behaves in different moments in time and if it exhibits any identifiable patterns. Such drawbacks make the study and analysis of churn in P2P systems difficult.

Most of the research in the area of churn in P2P systems is mainly focused towards some main factors: percentage of nodes being on-line at any time, session lengths, inter-arrival time of nodes (off-line time length) and overall availability. Works like the ones in [4, 18] focus on such parameters for different churn traces. Even though these works identify the general trend that there is a diurnal pattern in global peer population, they fail to relate these finding to individual peers. While the overall availability is interesting for the operation of an overlay, studying the availability of each individual peer is also useful, especially when in some overlay types, some of the peers have key roles. Such patterns are brought into attention by [32] where it is shown that nodes exhibit recurrent patterns in their on-line behavior.

It is shown in [29, 32, 38] that there is a need to study nodes individually for P2P systems in order to see that, besides the diurnal and weekly patterns in total number of nodes in the system, there are also individual patterns. These works call this a regular behavior of the nodes attributing it mainly due to the human nature and daily or weekly routines that make the same people do the same things at approximately the same time over and over again. What this goes to show is that churn has also a predictable side. However most of these works miss detailed investigation of the parameters that influence which subset of the nodes is considered regular, which is very important for applications that make use of the concept of regularity. Such an investigation is one of our goals in this thesis.

3.3 Using statistical information about nodes' availability

Churn analysis gives us a general idea of what to expect in a given P2P system. Some of the conclusions drawn from analyzing one system, might be generalized to apply to other P2P systems also. However, the results of the off-line churn analysis have not been traditionally used for improving the on-line behavior of protocols. That is why a few works have delved into the idea of how to predict nodes' availability based on statistical information on these nodes up and down times. Some of these works use concepts similar to the node regularity, while others fail to identify recurring patterns at the individual node level and therefore they do not take full advantage of this property.

3.3.1 Representation, exchange and on-line maintenance of node behavior information

One aspect of this research topic is how to gather and maintain information about other nodes in the system on the fly. This is inevitably related also to the method of prediction. In [28] the prediction method is a combination of saturating counter predictors, state-based predictors and linear predictors. The algorithm for predicting the next state of a node is a combination of decisions based on each of the predictors and then combining them for a final result. Nodes actively monitor each other and maintain the various predictors via this monitoring activity. However this on-line maintenance scheme for the predictors and also the predictors themselves seem to be quite complicated. The predictions that this method yields are only for short periods of time, meaning next couple of slots, which is not that useful for our case as we would possibly like for example to be able to predict an entire day or week of behavior. Even in the tests the prediction algorithm does not fare well in the conditions of a P2P system where nodes are managed by human users, such as Overnet. This makes it even less suitable for our purposes.

Another example, [25] uses a 7 day history composed of slots of 10 minutes where the state for each slots is recorded as it is. For each coming day, the prediction for a particular slot is "on-line" if the node was available for at least 5 (threshold) days in the history for that spot. This method of prediction of future behavior assumes that the nodes have a fixed daily pattern that can be identified by aggregating the behavior of 7 past days into a single pattern. Thus, as it is proposed in [25], this method cannot identify weekly patterns. In order to be able to support the identification of weekly patterns, for analogy to the approach proposed, the nodes would have to maintain a 7 week history instead of 7 days. Then, the prediction for each slot in the coming week would be "on-line" if the node was available for at least 5 (threshold) weeks in the history for that same spot. However, this is just an analogy that we are drawing on the method presented in [25] and it does not represent what is stated in that work.

The neighbor choosing in [25] is proposed to be based on the number of slots that two nodes have in common. The bigger this number (intersection of slot sets), the stronger the two nodes prefer each other as neighbors. Since the same highly available peers could be targeted as neighbors by most of the network, the authors have also added a concept of fairness by dividing the number of slots in common by the total number of slots where any of the two nodes is on-line (union of slot sets).

So to summarize, the approach proposed in [25] is far from optimal because it takes into consideration only diurnal patterns of the nodes, while omitting bigger patterns such as the weekly one. Also, keeping and possibly exchanging a history of every 10 minutes for a large number of nodes could put quite a bit of overhead in terms of traffic (a 7 day history has 1008 slots of 10 minutes), even though this depends, to a large extent, on the implementation. Another problem not discussed in this work is how the nodes gather information about other nodes and what information is exchanged between them.

A similar method to the one proposed in [25] for predicting future node behavior is also used in [12]. The work presents us with a number of different forms of predictors and their efficiency. These predictors can be classified as predictors that treat all the nodes the same without attributing individual properties to any of them and predictors that take into account individual node behavior. One predictor is a *flat* predictor which averages availability for all the timeslots and then the predicted value for any time in the future is this average. Other predictors are *Weekly periodic* which takes the average availability of nodes in the same time slot for every week; *Daily periodic* which takes the average availability for a time slot every day, without taking into consideration what day of the week it is; *Weekend-aware periodic* which takes into account the fact that the observed patterns of node behavior are largely different during weekends. The results from the tests conducted show that the accuracy is much higher when individual node behavior is taken into consideration and especially when weekly patterns and weekend patterns are considered. This goes to show that work-hours vs free-hours of the human users behind the nodes are important to be distinguished from each other, but also that the most prominent period of recurring behavior in humans is weekly.

Works in [37] and [39] have proposed a structure called a "peer availability table" (PAT), which is used to store information about peers that the node comes in contact with. The information that is kept in this table is the number of times the node was on-line and the total amount of times the node was probed, for each timeslot. This makes it possible to always have an average of the time the node has spent on-line in a specific timeslot, which compared to a threshold later determines whether the node is considered available in that spot in future occurrences of it. In order to keep the table size at check, the nodes for which the average availability has converged to 0 are removed. However, this data structure suffers from the problem of becoming bigger with time. Another problem again is that this method requires continuous probing of the other nodes in the system, which creates a lot of overhead in terms of overlay maintenance

messages. However, the biggest problem is how to make sure that all the nodes in the system have the same perception about the same node. This is not trivial since each node is maintaining its own PAT. These problems make this solution not very appropriate for P2P systems where bandwidth consumption and amount of overhead are important. It would also be almost impossible to have all the nodes in the system agree on a globally accepted value of the future availability of another node.

So the works mentioned in this section were proposing different ways of gathering and storing information about individual node availability and also various ways of how to use such information to predict future behavior of the nodes. We explained why many of the methods proposed in these works are not light weight and not entirely appropriate for P2P systems. We also came to the conclusion that the most widely accepted periods for which nodes exhibit recurring and predictable behavior are daily and weekly based. Moreover, predictors based on the weekly patterns of nodes are shown to be most accurate for predicting future node behavior.

3.3.2 Putting availability/regularity information to use

Gathering node availability information and predicting future node behavior is only part of the problem. It is also required to use this information in order to improve existing P2P solutions. This in great part depends on the application built on top of the P2P system.

The most prominent use of node availability information to predict future node behavior has been in improving data availability and bandwidth consumption for replica placement and replacement in data storage systems atop DHTs. Many different methods of doing such a thing are discussed in [32], which we mentioned in Section 1.2.3. The work in [32] also makes a survey of approaches for utilizing regularity information in the context of replica placement in DHTs. DHT data storage systems need multiple replicas in order to provide data availability in the face of a failure. Usually replicas are placed on nodes closest in the ID space to the node responsible for the data. However this might not be the most efficient in a dynamic system when nodes come and leave. The main problem is that this selection does not consider overall availability of the file over time, and the bandwidth usage due to data transfer when a replica that leaves the system has to be replaced by another one.

The simplest approach that tries to take advantage of node availability is assigning replicas to the nodes that have the highest availability in the near future. This is discussed in [29] and referenced in [32]. This method relies on availability predictors discussed in [29], but it is easy to adapt to the concept of regularity we are using in this thesis which is derived from the concept of regularity used in [32] and the concept of high availability in [40] and [38]. Using this concept, the method presented in [29] would be to assign replicas to nodes that are regular in more timeslots compared to the other nodes. Several possible improvements to this method are also discussed in [29] and [32].

Another approach proposed in [22] also referenced in [32] is placing replicas on nodes that are anti-correlated with respect to high availability periods. This approach ensures that at any time, there is a certain number of highly available nodes that hold the replica. This method of placing replicas is called anti-correlated. The concept of high availability in this paper is somewhat very similar to the one of regularity. The work proposes some heuristics to compare two vectors of regularity information of two nodes A and B and determine how well their patterns of availability are anti-correlated. A perfect anti-correlation would be when at any point in time exactly one node out of A and B is on-line. It is shown in [32] that anti-correlation is the most efficient way of placing replicas if overall bandwidth consumption is of concern. The intuition behind this result is that the purpose of a data storage system is to have the data available at any time, and preparing ahead of time makes sense as the best approach.

3.4 Conclusions

With regards to analysis of individual node behavior, there have been some attempts at identifying recurring patterns in their on-line behavior. Some of the works have focused on the same, or very similar, concept of regularity that we are using for this thesis. However, they have not gone into details about identifying the various parameters that influence which subset of the nodes is considered regular at various points in time. Therefore this is part of what we focus in Chapter 4.

While there have been attempts to use patterns of node availability in various research works, whether they call it regularity or not, no work has considered using such information in the context of publish-subscribe systems. The general tendency has been to use it in the context of DHT based storage systems to improve availability or lower bandwidth consumption for replica re-placement in periods of churn. No work has tried to use such information in other types of overlays or for other types of systems, such as we are trying to do with dissemination overlays for topic-based publish-subscribe.

None of the works described in [1, 2, 7, 8, 17, 20, 26, 34, 36, 51, 52] take advantage of the fact that nodes might show recurring behavior in their on-line and off-line behavior. They do not take into account that by analyzing statistical information about the nodes up and down times, future behavior of the nodes could be predicted, especially since behind real world nodes usually stands a human factor and humans show recurring temporal behavior. So in the case of designing overlays for message dissemination that could be targeted for applications that have humans as end-users, it makes sense to look into the possibility of using such information to aid with the overall efficiency of the dissemination. In the experiments some of these works conduct in the presence of churn, shows that their hit-ratio is not 100%. This means that there is room for improvement. That is why one of the goals of this thesis is to improve the hit-ratio of an existing P2P topic-based pub-sub system by using the concept of regularity.

Chapter 4

Regularity in existing systems

As we explained a little in the introduction chapter, it has been shown in previous research that nodes in P2P systems do not have a random behavior with regards to their on-line availability. In fact, nodes display some regularity in their connection patterns. In this chapter we explore the truthfulness of this statement and to what extent this is a reliable property and not just an exception. We try to identify the various parameters that influence the perception of nodes being regular. We also try to understand if nodes have the property of retaining the regularity attribute between adjacent timeslots. In order to do these we analyze availability traces from existing P2P systems.

4.1 Choice of traces for the analysis

Unfortunately, the availability of traces for P2P networks is limited. The research community has made an effort to provide data from real life P2P systems, however, because of the nature of our study, we require the traces to be over a relatively long time, at least more than 2 weeks in order to derive good results. In some way this confines the material available to us.

Some of the traces available are not of the required length or they are from systems that are well maintained and not from P2P systems where users are free to do as they please. Traces like Planetlab [43], Microsoft Facilities PCs [5], or many other traces available at http://fta.scem.uws.edu.au/ are from systems that are highly supervised and cannot offer unbiased information about regularity. Some other traces like Overnet [3], Gnutella [35] are not long enough to be considered for our purposes. Moreover, other traces might consider availability different from our concept, making them not useful in our case. For example, SETI@Home considers a node available when the CPU has free cycles, which is not related to the times when the user himself is using the service and therefore it is not what we are looking for.

Hence for this chapter we will analyze only the traces available from Skype super-peer network conducted in [18] and the trace from the KAD overlay network from the work in [42]. These traces will be explained in detail in Section 4.2.

4.2 Description of the traces

First we will describe how the traces were retrieved and what information they contain. We will talk a little about the format of the traces we are using and then we explain how we analyze the traces and in the end we show the results and conclusions of our analysis.

4.2.1 AVT - Availability trace format

Luckily, the Skype and KAD traces are in the AVT file format, which has become a popular format for availability traces of various distributed systems. Is is derived from the work in [35]. This is an easier to read and parse format derived from raw event files that are collected when nodes are sampled. The original raw format, as described in [23], is a list of events, for example node coming on-line or going off-line, represented in the form of a timestamp, unique node identifier and an event indicating available or not. These files tend to be very large and they are not very suitable for easy parsing. Therefore, researchers have come out with a more condensed version that is in the AVT format:

# Node ID	No. of Sessions	start-end timestamps tuples			
128.2.1.2	2	0.0	1.0	5.6	17
136.152.132.74	0				
127.0.0.1	1	500	10000		

This example shows a simple .avt file having the records of 3 nodes. In this specific case the nodes are identified by their IP, which is the first column in the file. The second column is a single integer number that represents the number of sessions that are present in the file for the corresponding node. A session is composed of two timestamps, the start of the session when the node comes on-line and the end of the session when either the node went off-line or the sampling for the trace stopped. So if in the second column we have the number n, then in the next column we expect to have $2n$ numbers representing timestamps. In the example file we can see the second node has no sessions, which means that it was sampled, but never appeared to be on-line. The first node has 2 sessions, one starting at 0 and ending at 1 second, and the second starting at 5.6 seconds and ending at 17. The timestamps usually represent seconds and are relative in the trace, starting from a timestamp of 0 which signifies the starting point of probing the nodes.

4.2.2 Skype trace

Skype is one of the most popular VoIP applications that have ever existed in the Internet. It was founded in 2003. Even though the protocols behind Skype are proprietary and nobody knows all the details how they work, there are some speculations that it is similar to the KaZaA network [16]. Different from most VoIP solutions, Skype takes advantage of a P2P underlying overlay in order to implement great parts of their protocol and

off-load some of the traffic during calls or instant messaging services to peers and also for NAT traversal and other functionalities. This is why traces from this solution are interesting for our study.

The trace we use for Skype comes from the study conducted in [18]. The study was made for the period September 1, 2005 to January 14,2006. Since then Skype has changed a lot and therefore the underlying protocols or details of it might have changed. What we explain concerns the traces and what was true at the time of the study. The reason for using such old traces is that newer ones are not available to the best of our knowledge.

The overlay network of Skype is based on the concept of super-nodes. All the nodes in the network are arranged in a two layers: super-nodes and normal nodes. Super-nodes are ordinary nodes, but selected to be promoted to super-node status on basis of some criteria. They have an overlay among each other and the general overlay that connects them also to the normal nodes. The normal nodes select one or a group of super-nodes to connect to and these are used in order to make queries on the network for finding information, routing call etc.

The availability trace that we use for Skype is a pinging of the super-node network in the period September to October 2005. The experiments for getting this trace were conducted in several phases, one of which was about exploring the Skype network and trying to find super-nodes and normal nodes. Next the availability of super-nodes was measured, since they are the only ones that the study was able to monitor. The study says that the trace has 6000 super-nodes in total, but only 2078 of them ever responded to pinging. This is also reflected in the downloaded trace[1]. The methodology of getting the trace is by taking snapshots of the super-peer network to see which nodes are on-line. Each snapshot takes 4 minutes to execute and then it is re-executed after 30 mins. Therefore the granularity for our purpose of studying regularity is 30 mins, even though the time between two consecutive pingings of the same node is between 30 and 34 minutes.

4.2.3 KAD trace

KAD is a P2P DHT routing protocol based on Kademlia [27]. Many applications make use of this type of overlay, such as Overnet, eMule etc. The eDonkey P2P file sharing network also makes use of KAD and this makes this overlay quite popular with a large user base. Similar to Pastry and other DHTs, KAD uses 128 bit long hashes as IDs for the nodes in the system.

The trace that we have was collected by the authors in [42]. They used a crawler they built themselves in order to go through nodes in the system and log their status, IP and KAD ID. The crawler tries to do a breadth first search of the network by asking peers for other peers they know and in this way trying to get to know as many peers as possible. In the study it

[1] from http://www.cs.illinois.edu/~pbg/availability/ and from http://fta.scem.uws.edu.au/

is mentioned that in this way they could achieve a full crawl of the KAD network, however this would generate an excessive amount of data and network traffic and they could not handle more than 3 crawls a day, which is not really helpful for studying the network. Therefore they decided to do only a **zone crawl**, exploring just a 8-bit zone, meaning that they crawl only KAD IDs that start with the same 8-bit string. In total, the space crawled is $\frac{1}{256}$ of the entire ID space of KAD. As a prefix for the crawl was chosen $0x5b$. The trace has 400375 nodes in it and this includes nodes that have at least one on-line session, whatever the length of it. The pinging of the nodes was done every 5 minutes for the period between 23 September 2006 and 21 March 2007.

4.3 Methodology of analysis

In our analysis of the traces presented above, we want to be able to identify the optimal parameters that should be used with regards to regularity. We are interested in analyzing the traces for different timeslot durations and for CRB durations of one day and one week. As mentioned in Section 2.3, other CRB durations do not make much sense to be used due to normal human cycles of day and night, or of weekdays/workdays and weekends. As for timeslot durations, we try different values: 15, 30, 60 minutes. We consider timeslot durations of greater length to be not very useful as then the data we gather becomes too attenuated and not very helpful. The regularity thresholds we are interested in are from 60% to 95%, every 5%. Regularity thresholds lower than 50% do not make sense because they mean that the node is 'regularly' on-line in that timeslot for less than half of the timeslot, which contradicts the meaning of regularity.

The granularity of time present in the traces is one second and therefore this is the granularity we also use for our analysis. For each node present in the traces, we go through all the on-line sessions in the trace about that node and do our computations. For each timeslot we maintain two values, the total number of seconds that appear in the trace about that specific timeslot and the number of seconds that the node has been on-line in that timeslot. For each session, we determine the timeslots it covers and add the appropriate amount of seconds to the on-line value of those timeslots The total ratio between this total number of seconds being on-line in a timeslot and the total number of seconds that appear in a timeslot, is the ratio that has to be compared with the regularity threshold γ. If this ratio is greater than γ then we say that the node is regular in that timeslot.

For each timeslot we can then create summaries of the total number of nodes that are regular in it. We are then going to try and see the relation between the length of the timeslot, the regularity threshold γ and the number of regular nodes that appear in each of the timeslots. We can then use this information to determine what values we can use for the different experiments we conduct in Chapter 6. This information could be used to find an appropriate ratio between the information we have to keep, and how helpful our predictions are. This ratio is application specific

and therefore it has to be determined by the application taking advantage of regularity. However, it should be noted for such applications that the higher is the number of timeslots, the higher is the memory consumption to keep track of them and the higher is the CPU usage. However, the longer a timeslot is, the more attenuated and unreliable the prediction of regularity for each specific moment in time is. The regularity threshold γ could be carefully chosen to balance between the number of nodes perceived as regular for each timeslot and the probability that they are going to be on-line at any moment in time during a timeslot they are regular.

For the analysis of the traces we have developed our own C++ application that goes through the file and processes the information according to the specified timeslot length, regularity threshold and CRB (day or week). The output is in the form of two columns: the timeslot number and the number of nodes that are considered regular for that timeslot. Another piece of software was developed in C++ in order to compute the presence of nodes in the system for both traces. This application goes through the seconds between the start and the end of the trace and goes through all the nodes' sessions to see which nodes are on-line at that specific time and which are not.

4.4 Results of trace analysis

The analysis is divided into the following parts:

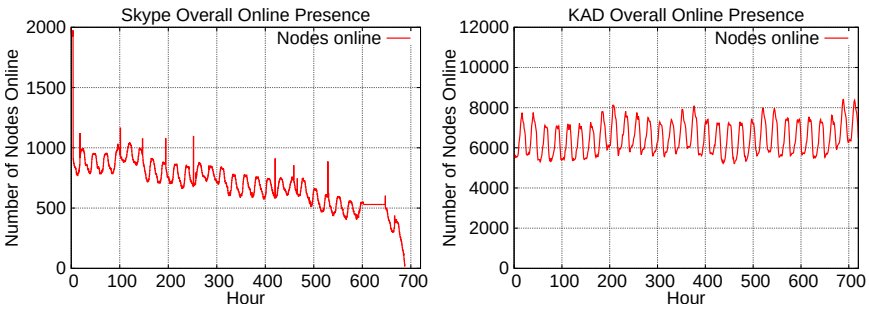
- The relationship between the *timeslot duration* and the number of regular nodes per timeslot. The measurements are conducted for different regularity thresholds and for daily and weekly CRBs.
- The relationship between the different *regularity thresholds* and the number of regular nodes in each timeslot.The measurements are conducted for different timeslot durations and for daily and weekly CRBs.
- For each timeslot t we want to see the percentage of nodes that continue to be regular in t+1 if they were regular in t. The measurements are conducted for different regularity thresholds and timeslot durations.

The results are presented in the form of graphs in the following sub-sections.

First of all, it is important to have a general overview of the information that the traces contain with regards to the total number of nodes that are on-line over the whole duration of the trace. For this reason we have plotted

Table 4.1: Average number of on-line nodes

Trace	Avg. Num. Nodes On-Line.
Skype	700
KAD	6997



(a) Skype superpeers on-line node presence (b) KAD network on-line node presence

Figure 4.1: Presence of nodes in both systems

the availability traces for Skype and KAD in Figure 4.1a and Figure 4.1b respectively. Since the KAD trace is quite long, we are presenting only part of its length for clarity in the plot.

Figure 4.1a shows that the Skype trace has some anomalies not explained by the authors. However, these anomalies are a good opportunity for researchers to use this trace in their simulations to test situations like flash crowds, as appears in the beginning of the trace, or massive node departure from the system as appears in the end of the trace. The total average number of nodes present on-line during the whole duration of the trace is 700.

Figure 4.1b shows that the KAD trace is more well formed. It has fewer abnormal artifacts and follows a more normal and intuitive behavior of the nodes according to night-day cycles and also weekly cycles. There are evident spikes in node participation in the system during two days of the week, which happen to be weekends, while the other days have lower participation. This shows that the overall node population exhibits a weekly pattern, while, for both traces, the daily patter is also visible since the plot follows a shape resembling a sinusoid. For the KAD trace, the average number of nodes that are on-line is 6997.

4.4.1 Relationship between timeslot length and number of regular nodes

At first we analyze the effect that the timeslot length has on the number of nodes that are perceived as regular in each of them for different regularity thresholds γ. Because the results were similar for all the γ values we tested, we are showing only the results for $\gamma = 60\%$ and $\gamma = 70\%$. The results were plotted and are presented in Figures 4.2, 4.3 for daily CRB and Figures 4.4, 4.5 for weekly CRB.

As it is seen from these graphs, the timeslot duration does not seem to influence the number of nodes that are perceived as regular in each of them. On the other hand, decreasing the duration of the timeslot gives us a better ''resolution'', which means that we can see with more detail the small changes in the number of regular nodes in each of the timeslots. This also means that the node downtimes in each of the timeslots are equally

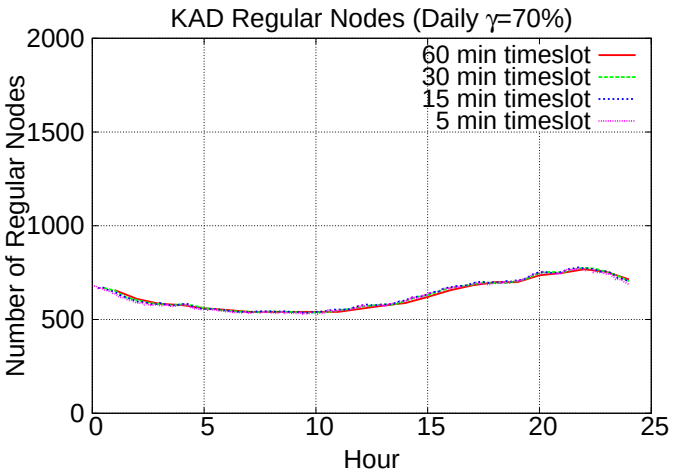


Figure 4.2: KAD daily regularity analysis with regularity threshold $\gamma = 70\%$ with different timeslot durations

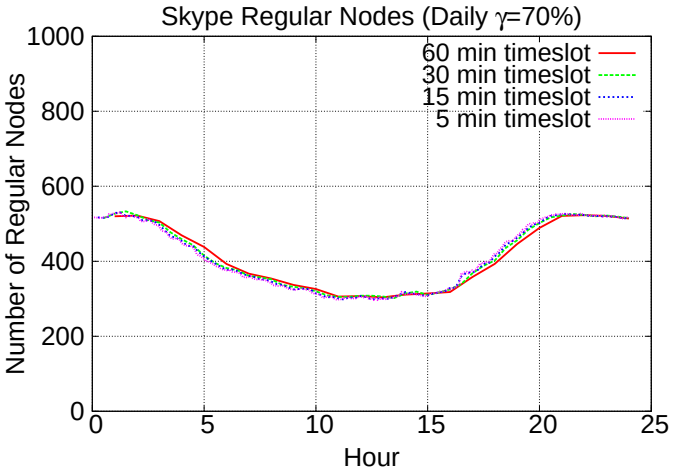


Figure 4.3: Skype daily regularity analysis with regularity threshold $\gamma = 70\%$ with different timeslot durations

distributed in the length of the timeslots. With the decreased duration of each timeslot the ratio of on-line time and off-line time changes very little since the number of regular nodes is almost not changing. This is especially helpful in showing that choosing a timeslot length of 60 min is practically no different in terms of number of regular nodes than any other timeslot length. This result is important for us and other researchers using the concept of regularity since it means that the timeslot duration does not affect the perceived number of regular nodes. For applications taking advantage of regularity, this finding means that it could be possible to save computation time and memory footprint since we need to store less data (less timeslots) for 60 min timeslots than with 30, 15 or shorter timeslots. These results are consistent with daily and weekly CRBs.

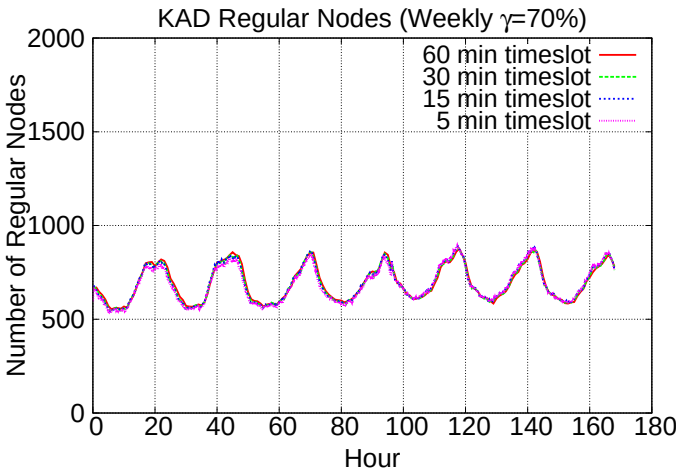


Figure 4.4: KAD weekly regularity analysis with regularity threshold $\gamma = 70\%$ with different timeslot durations

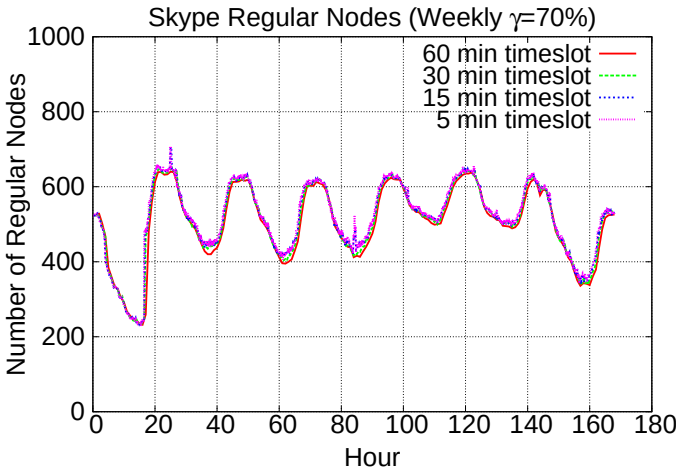


Figure 4.5: Skype weekly regularity analysis with regularity threshold $\gamma = 70\%$ with different timeslot durations

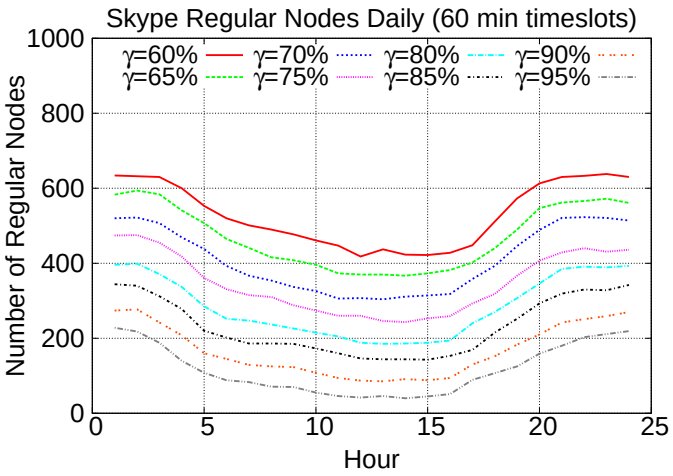


Figure 4.6: Skype daily regularity analysis with timeslot duration of 60 min and different regularity thresholds

4.4.2 Relationship between regularity threshold γ and the number of regular nodes

The other analysis that is meaningful for our purposes is the relation between the regularity threshold and the number of regular nodes that are perceived for each timeslot. The results were plotted and are presented in Figures 4.8, 4.6 for daily CRB and Figures 4.9, 4.7 for weekly CRB. The fact that there is going to be a lower number of nodes perceived as regular for each timeslot when we increase γ is obvious. What we would like to see is how much an increase in γ affects this number. Also it is important to see the number of regular nodes for different γ values in relation with the total number of nodes in the trace and the number of nodes on-line at any moment in time.

It is immediately evident that, even though the number of regular nodes for each timeslot is higher in KAD, the total number of nodes in the trace is also higher for KAD ($\approx$ 400000) and therefore, in percentage, the ratio of regular nodes is lower than in Skype.

The graphs show that there is a clear reduction in the number of regular nodes in each timeslot when the regularity threshold is increased. This is especially true for the daily CRB (Figures 4.8, 4.6) and also for KAD analysis of weekly CRB (Figure 4.9). For Skype, the weekly CRB graphs are a bit unpredicted because the values seem to be clustered together: the results for γ values of 60%, 65% and 70% and the results for γ values of 75%-95%. This result can only be explained by the fact that the nodes that were queried in the Skype trace were the super-nodes, which are selected according to some proprietary protocol that Skype implements and we do not actually know the details of how this choice is made. We could speculate that Skype chooses super-peers to be nodes that exhibit regularity in their availability patterns with a weekly repetition interval. This hypothesis is to some extent supported by the graph in Figure 4.7. The same result is also seen for other timeslot durations.

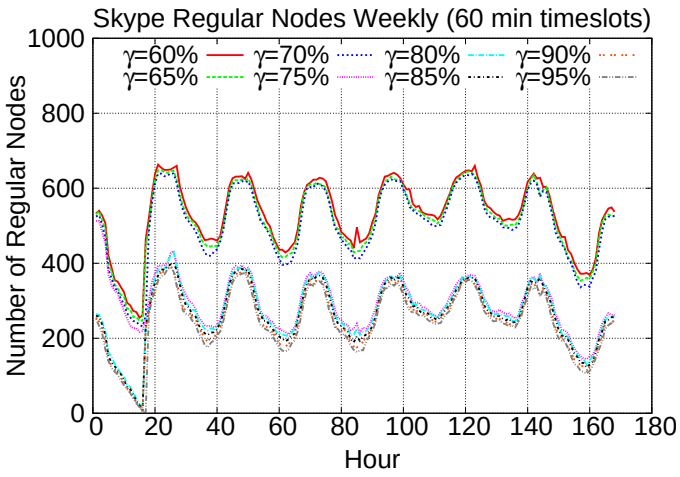


Figure 4.7: Skype weekly regularity analysis with timeslot duration of 60 min and different regularity thresholds

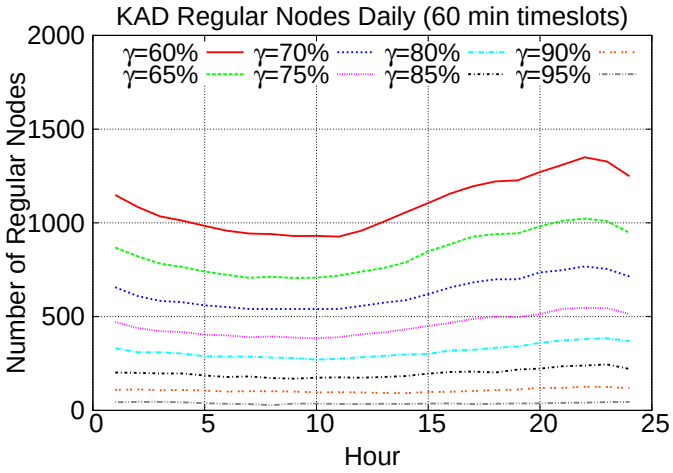


Figure 4.8: KAD daily regularity analysis with timeslot duration of 60 min and different regularity thresholds

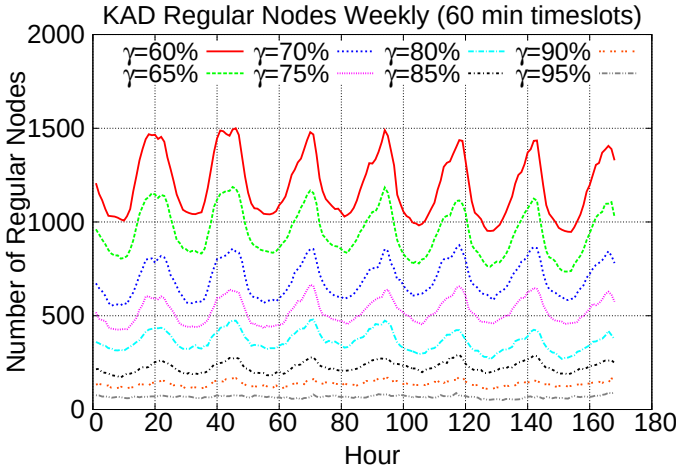


Figure 4.9: KAD weekly regularity analysis with timeslot duration of 60 min and different regularity thresholds

Table 4.2: Average Number of regular nodes for Skype for different regularity thresholds γ

γ	Avg. Num. Regular Nodes Daily Reg.	Avg. Num. Regular Nodes Weekly Reg.
60	532	521
65	475	520
70	409	518
75	349	284
80	283	261
85	238	260
90	164	259
95	120	258

Table 4.3: Average Number of regular nodes for KAD for different regularity thresholds γ

γ	Avg. Num. Regular Nodes Daily Reg.	Avg. Num. Regular Nodes Weekly Reg.
60	1095	1290
65	836	966
70	625	685
75	449	540
80	314	413
85	195	216
90	103	136
95	36	75

Average numbers of regular nodes per γ value are presented in Table 4.3 and Table 4.2.

4.4.3 Percentage of nodes retaining regularity between timeslots

The notion of regularity that we are using has a fixed duration for a timeslot and, immediately after a timeslot is over, the time of the next one starts, considering other nodes as regular for this one. As we will discuss more about it in Chapter 6, this could constitute a problem due to lack of synchronization in distributed systems. The nodes in the system will consider that the timeslot has changed based on their own clocks, which might not be in sync with the clocks of other nodes and therefore they will not perceive that the timeslot has changed. In case most of the nodes retain their regularity in-between contiguous timeslots, then the risks resulting from the lack of coordination can be minimized. Therefore we are interested in measuring this as a metric: the percentage of nodes that retain their regularity in-between contiguous timeslots.

To calculate this metric, we consider the set A of nodes regular in timeslot t and set B of nodes regular in timeslot t+1. The percentage of nodes that remain regular in t+1 when they are regular in t is therefore $\frac{|A \cup B|}{|A|} \cdot 100$. Typical results for a selected set of settings are presented in Table 4.4 and Table 4.5.

It is clearly visible from these tables that there is a drop in the percentage of nodes that retain regularity between timeslots with an increase in the regularity threshold γ. However this drop is quite low on the overall average. Mostly this drop affects the minimum value, so it is representative of the worst case scenarios that can occur.

Table 4.4: Daily CRB, percentage of nodes that retain regularity between timeslots

		Min	**Max**	**Avg**
KAD Daily 300 sec timeslot	γ**=60%**	97.59%	99.89%	99.16%
	γ**=70%**	97.65%	100.00%	99.02%
	γ**=80%**	95.74%	100.00%	98.83%
	γ**=90%**	93.04%	100.00%	98.12%
KAD Daily 3600 sec timeslot	γ**=60%**	88.15%	97.49%	95.22%
	γ**=70%**	88.10%	97.41%	94.93%
	γ**=80%**	85.64%	97.36%	94.19%
	γ**=90%**	84.00%	98.00%	92.37%
Skype Daily 300 sec timeslot	γ**=60%**	96.95%	100.00%	99.35%
	γ**=70%**	96.35%	100.00%	99.17%
	γ**=80%**	93.99%	100.00%	98.93%
	γ**=90%**	92.63%	100.00%	98.64%
Skype Daily 3600 sec timeslot	γ**=60%**	89.49%	98.83%	95.14%
	γ**=70%**	86.53%	99.69%	93.84%
	γ**=80%**	82.49%	98.47%	91.99%
	γ**=90%**	73.56%	98.54%	89.54%

Timeslot duration is also shown to affect the percentage of nodes that retain regularity. This is predictable since longer timeslots takes into consideration periods of time that include many shorter timeslots.

The overall conclusion is that with such averages as presented in these two tables, the churn in the population of regular nodes in-between contiguous timeslots is quite small. This means that on average more than 88% of the regular nodes continue to remain regular for the duration of at least two contiguous timeslots. This allows us to be more liberal with the assumption about the amount of clock drift between the different nodes in the system. We will have a longer discussion about this in Chapter 6.

Table 4.5: Weekly CRB, percentage of nodes that retain regularity between timeslots

		Min	Max	Avg
KAD Weekly 300 sec timeslot	γ**=60%**	80.44%	99.84%	98.82%
	γ**=70%**	77.91%	100.00%	98.68%
	γ**=80%**	75.51%	100.00%	98.53%
	γ**=90%**	62.75%	100.00%	97.95%
KAD Weekly 3600 sec timeslot	γ**=60%**	82.72%	96.60%	93.02%
	γ**=70%**	81.05%	97.38%	92.57%
	γ**=80%**	80.63%	96.93%	91.43%
	γ**=90%**	77.51%	97.44%	88.35%
Skype Weekly 300 sec timeslot	γ**=60%**	66.79%	100.00%	99.30%
	γ**=70%**	86.03%	100.00%	99.32%
	γ**=80%**	0.00%	100.00%	98.86%
	γ**=90%**	0.00%	100.00%	98.88%
Skype Weekly 3600 sec timeslot	γ**=60%**	80.87%	99.18%	94.26%
	γ**=70%**	79.06%	98.88%	94.06%
	γ**=80%**	40.00%	99.44%	90.72%
	γ**=90%**	29.41%	98.87%	90.06%

4.5 Conclusions

As it can be seen from the graphs presented in Section 4.4, there is a trade-off between the regularity threshold γ and the number of regular nodes that appear in each timeslot.

Regularity Threshold:

The higher γ is, the lower the number of nodes regular in each timeslot is, however the more reliable these nodes are (they have higher probability of being on-line during the timeslot).

With higher levels of γ, the nodes that are regular in a timeslot have a proportionally higher probability to be available in that timeslot and with less disconnections during the timespan of it.

The other conclusion that we can derive from our analysis of the traces is that:

Timeslot Length

The length of the timeslots has **no** perceivable effect on the number of nodes that are regular in each timeslot.

This allows us to select the timeslot length that better suits the implementation and performance of the application that is going to use this regularity information. However, one should keep in mind that the timeslot duration should not be so long as to loose meaningfulness to the application that will use the regularity information. If for an application it makes sense to make a periodic decision based on regularity information,

then that period duration could very well be the duration of the timeslot. This means that if decisions need to be made every 20 minutes, it is not very helpful to have a timeslot of 120 minutes as there will be no change in the available information for 5 decision cycles.

From a broader view, we can argue that, by comparing Table 4.1 with the Tables 4.3 and 4.2, the ratio between regular nodes and normal nodes, is higher in the Skype trace (on average). For Skype the percentage of regular nodes is around 30%, varying between 17%-76% for the daily regularity and between 37%-74% for the weekly one.

On the other hand, for KAD we could say the opposite. The number of regular nodes is lower relative to the total number of on-line nodes. Here, the percentage of regular nodes is varying from .05%-15% for the daily regularity and between 1%-18% for the weekly one. This means that the ratio between regular nodes and the overall population of on-line nodes at any moment in time is low.

This means that, in our opinion:

Availability Trace To Be Used

The Skype trace is a more suitable trace for regularity experiments.

Therefore we will use this trace later in Chapter 6 to try out our new approach for improving dissemination in PolderCast using regularity.

We were also interested to see how nodes retain the regularity property between consecutive timeslots. This was in order to see how big the churn in the regular nodes population was. This is important to discuss how a system that takes advantage of node regularity would behave in the real world where clock drift is an unavoidable issue which could make nodes perceive they are in different timeslots, especially in the short timespan in-between timeslots. The conclusion is that:

Regularity Retention

Over 88% of the nodes retain their regularity in timeslot **t+1** if they are regular in timeslot **t** for all the settings we tried, showing that churn in the regular population is $\leq$ 12%.

Chapter 5

GRID: A generic regular nodes discovery service

In this chapter we will introduce GRID (Generic Regularity Identification and Discovery Service) which is an attempt to create a generic service for keeping track of a node's own regularity and searching nodes across the network that are regular in a given timeslot.

5.1 Motivation: The need for a generic service

As we showed in Section 3.3.2, the regularity information is being used to a good extent in various types of systems and our work is trying to expand even more on the range of systems using this concept. This creates the need for a generic system that is able to find regular nodes in a given timeslot across the network. Various systems might use such a service in order to improve their inner workings to achieve better results for their specific purposes.

With such a service created, it is possible to focus on finding new ways of using the concept of regularity to improve existing systems or possibly new types of overlays.

5.2 Functionality: Goals of the service

The service that we are presenting has the following functionality:

- Keep track of the node's own regularity pattern.
- Find other regular nodes in the system for a particular timeslot.
- Perform application specific distributed clustering where nodes with similar regularity patterns will be clustered together.

These functional requirements come from the general goal to provide a service which is generic enough to benefit applications that want to take advantage of node regularity as we have defined it in this thesis. The third point in the functionalities of the service is especially aimed at fulfilling

this goal. The service interface defines what information the application needs to provide to the service through a callback and what information it can obtain from it. The details of the callback implementation are left to the application developer. It is through this callback that the application developer influences the distributed clustering performed by this service. The interface of GRID is presented in Section 5.4.

The service is distributed and there is a local agent of the service running on each machine where the application is deployed.

5.3 Design

The internal architecture of the service and how it interacts with GRID's instances on other nodes and with the application using it, is presented in Figure 5.1. While designing GRID we chose a layered approach, which we are briefly explaining in this overview:

- Cyclon - a gossiping protocol that serves the purpose of a random peer-sampling service.
- Victor - a gossiping protocol that provides the functionality of distributed clustering customizable by the application developer. To fulfill its goals, it uses nodes from the view of Cyclon, while at the same time gossiping with other instances of Victor on other nodes running GRID.
- Own Regularity Identification Service (Regularity Service for short) - a local service that identifies the regularity pattern of the node running GRID. The regularity pattern is converted into a bit-array of 1 bit per timeslot and a value of 1 means that the node is regular in that timeslot while 0 means that contrary. This information is included into the descriptor used by Cyclon and Victor as described by the Data Structure 1 into *regularityInformation*.

We will explain more about the functionality of each of these functional parts in Section 5.5.

The basic concept behind the service is the identification of the node's own regularity pattern and advertising it in the node's descriptor (shown in Data Structure 1) in the form of a bit string of 1-s and 0-s. These bits represent the state of regularity per each timeslot, where, intuitively, 1 means the node is regular in that timeslot and 0 means that it is not. This functionality is provided by the Regularity Service component of GRID. The inner workings of this service component are presented in Section 5.5.1.

The service that we are designing needs to be self sufficient. It has to be able to find regular nodes across the whole network over time in order to best serve the needs of the application using it. As suggested by the results in [46] and other similar works, it is shown through simulations that clustering algorithms converge faster and are able to function better under churn when combined with a random peer-sampling protocol from which

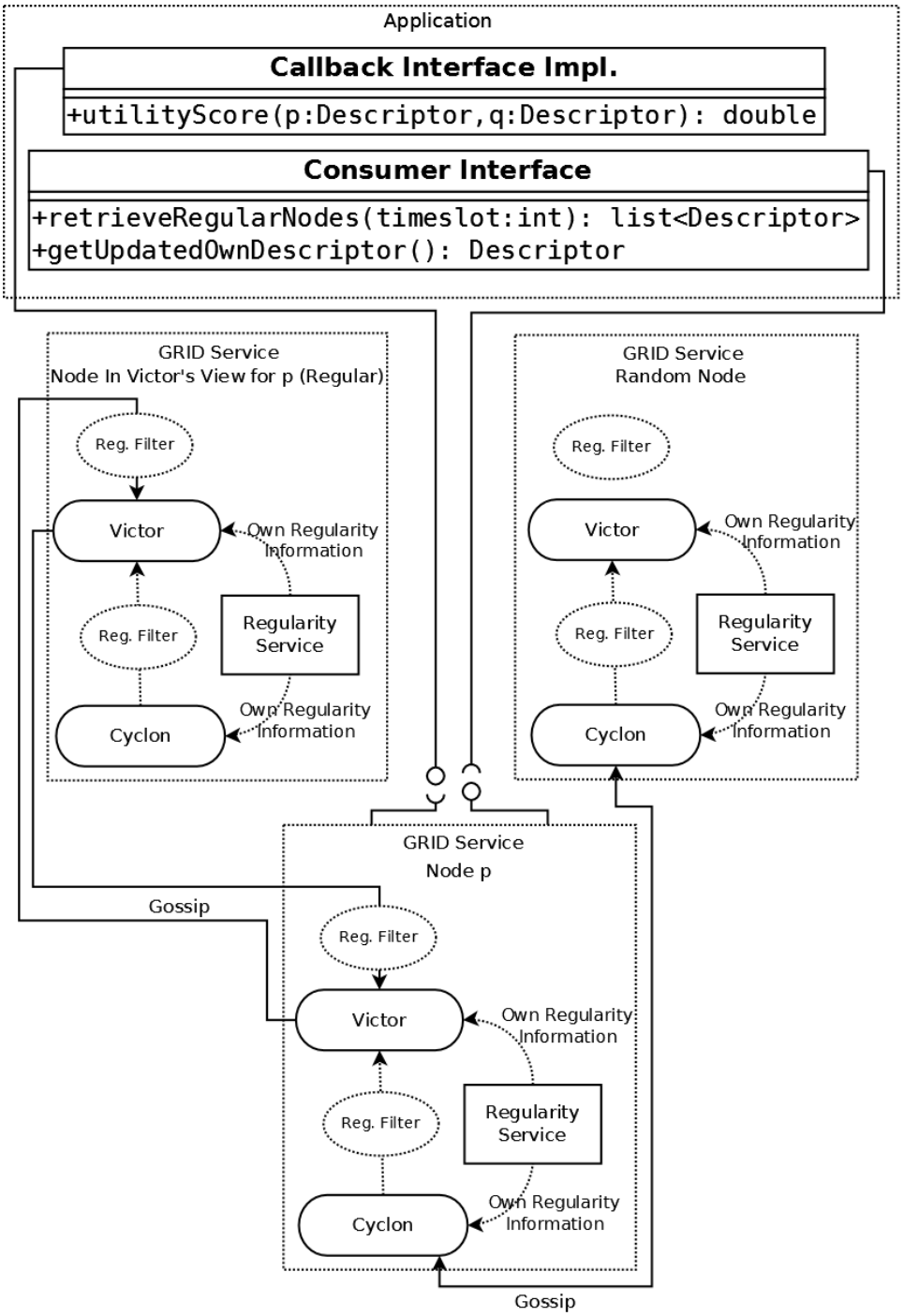


Figure 5.1: The diagram of the architecture and interactions of the service with the application and between instances of the service on different nodes

Data Structure 1 Data structure of a *descriptor* of a node

Descriptor(ID, info, age, regularityInformation)
Where:

- *ID* is the node ID
- *info* is the node's contact information like IP, Port etc.
- *age* is a number representing the age of the descriptor, showing how long it has since being communicated with
- *regularityInformation* is a bit string with one bit per timeslot and the bit is set for the timeslots where the node is regular.

they can retrieve nodes that come from various parts of the overlay. This is because random networks are proven to have high connectivity, while clustering concentrates connections between specific groups of nodes. Such reasoning is also used in [8] behind the *random coverage heuristics*.

For this reason at the base of the service we have put a random peer sampling service such as Cyclon (explained in Section 2.4.1). The purpose of this module is to always provide a random sample of the nodes in the network that is constantly changing over time. This random sample aids in finding gossiping partners for the layer above it.

The layer above Cyclon is called Victor (**Vic**inity ex**t**ended f**o**r **r**egularity) and it is used to offer distributed clustering of nodes with similar regularity pattern and application specific clustering. Victor, as we will explain in more detail in Section 5.5.3, has a view V_{Victor} made up of buckets, each of them being a set of descriptors. There is one bucket for every timeslot and each of them has a maximum capacity of l_{Victor}. V_{Victor} is also defined in Data Structure 2. For each of these buckets, the goal is to cluster nodes that are regular in the corresponding timeslot. On the other hand, the only bucket where Victor instances are actively gossiping with each-other is the one that corresponds to the current timeslot (the timeslot corresponding to the current time).

Cyclon only delivers a view consisting of random nodes across the network, while Victor needs nodes that are regular in the *current timeslot*. That is why, when the view of Cyclon is consumed by Victor, we need to **filter** only such nodes. Only nodes that are regular in the current timeslot can help improve the view of the current timeslot in Victor. This same filter is also applied when a node receives a subset of another node's view during gossiping. It can also be seen in Figure 5.1 where this filter is situated (i.e. the **Regularity Filter**).

In an ideal situation there should not be any need to filter out nodes being gossiped between two Victor instances since they are both gossiping on nodes regular for the current timeslot. However, as we explained in Section 4.4.3, in practice there are problems with clock synchronization in distributed systems that could cause different nodes to think that they are

in different timeslots. The filter we just explained makes sure that, when a node p is sending a set of nodes during gossiping to another node q and the clocks of p and q are slightly out of sync, only those nodes that are regular in the timeslot perceived by q as the current timeslot are going to be considered for improving the view of q. According to the analysis we conducted in Section 4.4.3, the effect of this filter during the change in timeslots is merely going to discard on average $\leq$ 12% of the nodes of this gossip exchange of neighbors. This happens because node q will not consider these nodes as regular in the timeslot q thinks is in. In Section 4.4.3 we showed that, for all the various regularity settings we tried, the average number of nodes that retained their regularity in timeslot t when they were regular in timeslot t-1, was $\geq$ 88%.

We have decided for GRID to offer a **callback interface**, the implementation of which is provided by the developer of the application that is going to be the consumer of GRID. This callback interface requires the implementation of a function **utilityScore** that assigns a score to a node's descriptor when compared to another node descriptor used as a reference. This function is then used in a sorting algorithm inside Victor in order to select the best nodes to keep into view when there are more nodes than the view size. The details of how and where **utilityScore** is used inside of Victor are explained in Section 5.5.3.

The implementation of the **utilityScore** function gives the application developer a way to modify the clustering that GRID performs in Victor. For example one implementation of **utilityScore** tailored towards topic-based publish-subscribed systems might cluster nodes according to the number of topics they have in common. In this case, the higher the number of topics the nodes passed to the function as arguments have in common, the higher the return value of the function. Another implementation of **utilityScore** tailored towards distributed file storage systems might cluster nodes according to the amount of free storage they have. In this case the function called on two nodes p and q would return a score proportional to the amount of storage node q has left. As can be seen from these examples, the application developer is free to give whatever meaning to the value returned by **utilityScore** and to tailor its implementation to the specific needs of the application using GRID. A more detailed example of how GRID can be configured by an application that wants to benefit from it is shown in Section 6.4.2.

The design of GRID makes it so that only the code inside the callback interface implementation has to be aware of application specific notions such as topics, topic coverage in the case of topic-based pub-sub systems, replica placement or amount of storage consumed in distributed file storage etc., while GRID just needs to have the knowledge about regularity.

In Section 5.5 we provide a more detailed view of the implementation of the different parts of GRID.

5.4 Interface to the service

As is shown also in Figure 5.1, GRID interacts with the application that is using it via two interfaces. The first one is the **callback interface** that needs to be implemented by the application using GRID. The second one is the **consumer interface** that the application can invoke to get information from GRID.

The interface that the application developer needs to implement and provide to GRID (i.e. the **callback interface**) includes only one method which is *utilityScore* that accepts two node descriptors, a reference node p and the node q to be evaluated. The descriptors are an extensible subclass of the basic descriptor in Data Structure 1, meaning that they can add whatever information they deem necessary for the application. The return value of the method is a utility score, which is used internally in Victor for sorting the descriptors compared to utility they bring to p. The higher the return value, the more preferred the descriptor q is for p. The details of how the *utilityScore* function is used inside Victor are presented in Section 5.5.3. It should be emphasized that, due to the **Regularity Filter** that is applied before nodes are ever considered for inclusion in the view of Victor, node q is always a node that is regular in the current timeslot.

The reason why the signature of the *utilityScore* function has 2 descriptors as arguments, is because this same function is used both when a node tries to improve its own view and when a node tries to determine which nodes would be best to send to a gossiping partner. So, when utilityScore(p,q) is called, node p can either be the local node, or a gossiping partner. If desired, the application developer can apply different score calculation when p is identified to be the local node as opposed to the case when p is any other node in the system. *utilityScore* is invoked in Algorithm 5. The two cases of using utilityScore occur when Algorithm 5 is invoked in Algorithm 3 and 4, respectively.

The information that the service offers is delivered via two methods of the **consumer interface**. One method is *retrieveRegularNodes(timeslot t)* which returns $V_{Victor}[t]$ (see Data Structure 2). $V_{Victor}[t]$ for a node p is the set of nodes q in Victor's view that are regular in timeslot t and that have had the best score returned from utilityScore(p,q), $\forall q$ Victor has encountered during gossiping or $\forall q$ Victor has retrieved from Cyclon's view. This is the ultimate purpose of the service. In case an application needs information for multiple timeslots, it can iterate and issue a request for each of them.

The other offered method in the **consumer interface** is *getUpdatedOwnDescriptor* which returns an updated version of the own descriptor of the node as presented in Data Structure 1, or an extended descriptor if the application opts to extend it. The updated piece of information in the descriptor is the node's own regularity pattern of the node running the service.

5.5 Implementation

In this section we describe in detail each of the components shown in Figure 5.1 and introduced in Section 5.3.

5.5.1 Own Regularity identification service

The regularity information of nodes is derived from the statistical information about the nodes' on-line and off-line times, by calculating a ratio of the two and comparing this ratio with the *regularity threshold*. First of all we should think about how to detect the regularity patterns of the nodes. One method could be to have nodes periodically exchange heartbeat messages with each other in order to detect when the nodes in their neighborhood are on-line. However, it would be impossible for nodes to accurately keep track of all the other nodes in the system. Moreover, it would be especially difficult to differentiate between a node being off-line and a node not being in the neighborhood. Another aspect to think about if going this way is how to make nodes exchange this information with each other in such a way that all the other nodes agree on the status of a node as being regular or not in a particular timeslot. This is without doubt not an easy task. Most importantly, all of these operations would put a very high overhead on the nodes in terms of messages and amount of data exchanged. Therefore this approach would not be suitable for a large scale P2P system.

As an alternative to the method presented above, we make the assumption that nodes are able to keep track of their own on-line and off-line times, even when the node crashes (ungraceful leave). This is certainly a fair assumption to be made since the only thing needed for this is that nodes keep track of the times when they are started and the time when they go off-line. We presume therefore that nodes keep logs in which they:

- Write the time they come on-line. An example message can be "ON <*timestamp*>" (coming on-line at *timestamp*).
- Periodically write the time to the log while they are on-line (e.g. write "BO <*timestamp*>" for being on-line at *timestamp*).
- Write the time they go off-line. (e.g. write "OFF <timestamp>" for going off-line at *timestamp*)

This way they can also keep track of the down times in the event of ungraceful leaves: When coming on-line after an ungraceful leave (i.e. the last message in the log is not an "OFF" message), the node reads from the log the timestamp of the last "BO" message and presumes that the time passed since this timestamp has been off-line.

Based on this temporal information the nodes calculate the total amount of time passed on-line and off-line for each of the timeslots (according to the specified parameters of *timeslot* duration and *CRB* duration) and then calculate the ratio. Comparing if the ratio is greater than the *regularity threshold*, one gets a series of **true** or **false** values that can be very well converted into a bit string. This bit string carries all of the regularity

information of the node with very little overhead in terms of memory for the nodes and bandwidth when this information is shared with other nodes. For example, it could take as little as 168 bit = 21 bytes to store the information about a week of 60 minutes timeslots.

The calculation of the regularity pattern has to be done once every time a timeslot has elapsed and each time the node gets on-line in order to have an up-to-date regularity information. The calculation is not expensive in terms of computation or memory and therefore can be executed every time each timeslot has elapsed.

This makes it possible for us to have nodes publish their own regularity information together with their *descriptor*, as described in Data Structure 1. The application developer could then extend this descriptor in order to add application specific information into it, which can be used in the implementation of the callback interface.

The *getUpdatedOwnDescriptor()* method offered in GRID's consumer interface is implemented in the own regularity identification service. This method returns a fresh copy of the descriptor of the local node where the *regularityInformation* part of it is updated with the latest regularity pattern identified by the own regularity identification service. The method *getUpdatedOwnDescriptor()* is also called internally in the algorithms of Victor as we will see in Section 5.5.3. We are omitting the actual implementation of this method as it is trivial: It simply creates a new instance of a node descriptor (i.e. Data Structure 1) with all the information about the local node (i.e. ID, info), an age=0 and it copies the local node's regularity pattern, as identified by the own regularity identification service, into the field regularityInformation of the descriptor; This descriptor instance is returned by the method.

5.5.2 Peer Sampling service

As we explained in Section 5.3, at the base of our service is a random peer sampling service that follows the exact protocol as Cyclon [47], with the only difference being that the descriptors of the nodes are different from the ones presented in the work where Cyclon was introduced. Since the algorithm is identical to the one presented in [47], we are not going to present it here.

5.5.3 Victor: Generic clustering service by extending Vicinity

The Victor layer is an adaptation of the Vicinity protocol explained in Section 2.4.2, that uses the regularity information of the node it is residing on and other nodes in the system. In this protocol, being executed on a node p, there is a view V_{Victor}, presented in Data Structure 2, composed of buckets (sets of descriptors) of a maximum capacity of l_{Victor}, one per timeslot. Nodes gossip with each other exchanging at most g_{Victor} descriptors of nodes (Data Structure 1).

There is only one active bucket in the view at a time, which is the one corresponding to the current timeslot and we try to identify nodes that are

Data Structure 2 Data structure for the view of Victor

$Descriptor[totalNumberOfTimeslots][l_{Victor}]$ V_{Victor}

Where:

- *totalNumberOfTimeslots* is the number of timeslots in the CRB.
- l_{Victor} is the maximum capacity of the set of descriptors for each timeslot.
- $V_{Victor}[t]$ is a set of descriptors for timeslot t (a.k.a bucket for t). There are *totalNumberOfTimeslots* such buckets and each with a maximum capacity of l_{Victor}.

Algorithm 3: nextCycle algorithm for Victor

```
/* Algorithm executed every gossiping cycle of Victor    */
```
Data: considerSet[] - an array of sets of nodes, one set per timeslot.

1 currentTimeslot $\leftarrow \frac{simulationTime}{numSecondsPerTimeslot}$ **mod** totalNumberOfTimeslots;

```
/* Get fresh descriptor of the local node (called node 'p'
   for simplicity).  It contains updated
   regularityInformation from the Regularity Service and
   age=0.  It is the same function provided in GRID's
   consumer interface and explained in Section 5.5.1     */
```
2 $p \leftarrow$ getUpdatedOwnDescriptor();

3 $V \leftarrow \{x | x \in$ $\{V_{Cyclon} \cup V_{Victor}$[currentTimeslot] $\cup$ considerSet[currentTimeslot]$\} \wedge$ x.regularityInformation[currentTimeslot] = 1 $\}$;

```
/* Add own descriptor to the list                        */
```
4 $V \leftarrow V \cup \{p\}$;

5 considerSet[currentTimeslot] $\leftarrow \emptyset$;

```
/* Increase age of all neighbors                         */
```
6 **forall the** *descriptor* $\in V_{Victor}[currentTimeslot]$ **do**

7 descriptor.age $\leftarrow$ descriptor.age +1;

```
/* Select oldest neighbor to gossip with                 */
```
8 q $\leftarrow$ Select oldest neighbor in V_{Victor}[currentTimeslot][0];

9 q.age $\leftarrow$ 0;

10 neighborsToSend $\leftarrow$ selectBestNeighbors(q, V, NULL, g_{Victor});

11 Send *neighborsToSend* to q;

12 $V_q \leftarrow$ Receive from q a similar list of neighbors;

13 $V \leftarrow V \cup V_q$;

14 V_{Victor}[currentTimeslot] $\leftarrow$ selectBestNeighbors(p, V, NULL, l_{Victor});

regular in the current timeslot. The buckets for the other timeslots are not active, meaning that there is no gossiping with the nodes regular in those timeslots (unless they are also regular in the current timeslot and therefore part of the active bucket). In those buckets we just keep the descriptors so that the next time the time of the corresponding timeslot comes, we have a pre-built view instead of having to construct one from scratch. Even though it is likely that a past view for this slot is outdated, the changes are typically incremental because the regularity of a node does not change often and relatively many occurrences of the CRB are needed (i.e. for CRB with 1 day length it means that several days are needed) for it to change .

The selection of nodes in the active view is made so that we get nodes that are regular in the current timeslot and that have the highest score according to the implementation of the function **utilityScore** in the *callback interface*.

This protocol has both a cyclic element and a reactive one, meaning that it executes at fixed recurring points in time but also responds to gossip requests from other nodes executing the same protocol. The cyclic procedure executed at every gossiping cycle is the one presented in Algorithm 3. The procedure that responds to gossip requests from Victor instances running on other nodes, is presented in Algorithm 4. The execution of the protocol follows a similar algorithm to that of Vicinity (presented in Algorithm 1). The difference is that we filter the nodes coming from the Cyclon layer and from gossiping with other nodes to keep only nodes that are regular into the current timeslot, prior to applying the selection function on them. This filtering is represented by the Regularity Filter in the architecture diagram of GRID in Figure 5.1. Another difference between Victor and Vicinity is the fact that the view of Victor is designed so that there is one different active view for each timeslot. This can be seen also in the structure of V_{Victor} in Data Structure 2. There are also other differences which will be evident when we explain the various algorithms that are executed in the Victor protocol in the following paragraphs.

The main algorithm for Victor, which executes every gossiping cycle on an arbitrary node 'p', is presented in Algorithm 3. First of all, the information each node keeps about other nodes is stored in *descriptors*. This data-structure, represented in Data Structure 1, is also the one that is exchanged between the nodes during gossiping. The first step is to determine the index of the bucket in the view corresponding to the current timeslot, which is the timeslot that corresponds to the current moment in time. This is done by performing a simple calculation of the simulation time (or real time if it is a real implementation) divided by the number of seconds per timeslot (duration of timeslot), translated into a timeslot number by modulus of the total number of timeslots. This operation is shown in line 1 of Algorithm 3. The next step is to gather a list V of all the nodes present in the Victor view and Cyclon view that fulfill the condition of being regular in the current timeslot. We increase the age of all these nodes and select the oldest one to gossip with, node q. We reset the age of q to 0 and select the best g_{Victor} neighbors to send to q by calling *selectBestNeighbors* (Algorithm 5) on V with q as a value for *referenceNode*.

Algorithm 4: respondToRequest(Descriptor q, Descriptor[] receivedList)

```
/* This is the method executed when a gossip request is
   received.  It selects the neighbors most appropriate for
   the requesting node and sends them.                    */
```

Input: Descriptor q - the descriptor of the node we are receiving the request from

Input: Descriptor[] receivedList - the list of neighbors received in this request

1 currentTimeslot $\leftarrow \frac{simulationTime}{numSecondsPerTimeslot}$ **mod** totalNumberOfTimeslots;

```
/* Get fresh descriptor of the local node (called node 'p'
   for simplicity).  It contains updated
   regularityInformation from the Regularity Service and
   age=0.  It is the same function provided in GRID's
   consumer interface and explained in Section 5.5.1      */
```

2 $p \leftarrow$ getUpdatedOwnDescriptor();

3 $V \leftarrow \{x | x \in \{V_{Cyclon} \cup V_{Victor}[\text{currentTimeslot}]\} \wedge$ x.regularityInformation[currentTimeslot] = 1 }

```
/* Add own descriptor to the list                         */
```

4 $V \leftarrow V \cup \{p\}$;

5 neighborsToSend $\leftarrow$ selectBestNeighbors(q, V, receivedList, g_{Victor});

6 Send *neighborsToSend* to q;

7 $V \leftarrow V \cup receivedList$;

8 V_{Victor}[currentTimeslot] $\leftarrow$ selectBestNeighbors(p,V,NULL,l_{Victor});

In the same way, a list is received from q and merged with V. Then we reconstruct V_{Victor} by applying *selectBestNeighbors* on V, this time having p as the value for *referenceNode*.

The descriptors in V are also put into the buckets of the view for the timeslots where they are regular in. This is done in order for these descriptors to be considered for those timeslots when the times of those timeslots comes. This is achieved by putting the descriptors of the nodes in the buckets of *considerSet* data structure that correspond to the timeslots where they are regular in. When the time of a timeslot t comes (i.e. t becomes the current timeslot), the nodes in $considerSet[\mathrm{t}]$ are considered for inclusion into $V_{\mathrm{Victor}}[\mathrm{currrentTimeslot}]$ and the corresponding $considerSet[\mathrm{t}]$ is emptied. So practically these extra nodes are considered the first time the *nextCycle* procedure is called for a particular timeslot. This is done in order to improve the pool of descriptors available for future timeslots, or even try to have descriptors available for timeslots where the node is not on-line itself. The interaction with the *considerSet* variable is shown in lines 8-9 in Algorithm 5 where we see how *considerSet* is populated. Also we see how the values inside *considerSet* are used in lines 3 and 5 of Algorithm 3.

The process of considering nodes in considerSet for the corresponding timeslots is also executed every-time a CRB is over in order take into account consideration of nodes also for the timeslots where the node itself is not on-line (i.e. therefore GRID is not running on those timeslots). This is simply achieved by cycling through all timeslots t, and for each of them executing $V_{Victor}[\mathrm{t}] \leftarrow \mathrm{selectBestNeighbors(p,}V_{Victor}[\mathrm{t}] \cup considerSet[t]\mathrm{,NULL,}l_{Victor}\mathrm{)}$.

The selection function described in Algorithm 5 is a sorting algorithm that uses a comparing function described in Algorithm 6, which in turn makes use of the proximity function described in Algorithm 7. Every time there are more nodes available to make a selection upon (because of the nodes received during gossiping or the regular nodes from Cyclon + nodes currently in V_{Victor}) than l_{Victor}, the list of nodes is sorted according to the selection function described above, and then only the topmost l_{Victor} are kept, while the rest are discarded.

Algorithm 5: *Descriptor*[] selectBestNeighbors(Descriptor referenceNode, Descriptor[] neighborList, Descriptor[] excludeList, int numToChoose)

```
/* This function gives the best numToChoose neighbors from
   neighborList for the node referenceNode                    */
```

Input: Descriptor referenceNode - the node for which we are going to choose the best neighbors

Input: Descriptor[] neighborList - list of neighbors from which to choose

Input: Descriptor[] excludeList - descriptor to avoid in the final result

Input: int numToChoose - number of neighbors to return

Output: Descriptor[] returnList - list of best *numToChoose* neighbors from *neighborList* for the node *referenceNode*

```
1  Descriptor[] returnList;
2  if numToChoose< l_Victor then
3      sort(referenceNode, neighborList) using compare(...) function;
       /* Sorting is performed by using timsort, the default
          sort for Java and as a comparator the function
          compare(referenceNode, a, b) where a, b are the nodes
          to be compared                                       */
4  for i=0 to |neighborList| − 1 do
5      descriptor ← neighborList[i];
6      if descriptor = referenceNode then
7          continue to next descriptor;
8      forall the {t ∈ descriptor.regularityInformation :
       descriptor.regularityInformation[t]=1 } do
9          considerSet[t] ← considerSet[t] ∪{descriptor};
           /* Put the descriptor for consideration in the bucket
              of the timeslot where it is regular              */
10     if descriptor ∉ excludeList then
11         returnList.add(descriptor);
12         numToChoose ← numToChoose - 1;
13     if numToChoose = 0 then
14         break loop;
15 return returnList;
```

Algorithm 6: int compare(Descriptor reference, Descriptor a, Descriptor b)

```
/* This function compares two nodes a and b with regards to
   utility they bring to node reference. It returns a value
   < 0 when a is preferred, > 0 when b is preferred and 0
   when both are equal                                      */
```

Input: Descriptor reference - the reference node. The utility of the other descriptors is measured towards this node

Input: Descriptor a, Descriptor b - the nodes to be compared with regards to the utility they give to node *referece*

Output: 0 - when both a and b give equal utility, +1 - when b is prefered and -1 when a is prefered

```
/* Use the callback interface implementation provided by
   the application for utilityScore                         */
```

```
1 scoreA ← utilityScore(reference, a);
2 scoreB ← utilityScore(reference, b);
3 if scoreA = scoreB then
4     return 0;
5 else
6     if scoreA > scoreB then
7         return −1;
8     else
9         return +1;
```

5.6 Limitations

GRID aims, among other things, to find regular nodes for all of the possible timeslots. However, it is not guaranteed that the service will find regular nodes for timeslots where the node has never been on-line itself, because of the way the service is designed. The reason for this problem is explained below in this section. Therefore our service is limited in its functionality as it is not able to serve those applications that require accurate information about regular nodes in all of the timeslots.

Such a case would be the one of finding nodes that have a regularity pattern that is anti-correlated to that of the current node. The concept of anti-correlation is explained in [22] and [32] and in the case of GRID it means that, if node p is a node running GRID, then it is trying to find nodes that are regular on every timeslot that node p is not. A simple example that is given in [22] is also presented in Figure 5.2. However, since node p is not running the service in such timeslots, it might not be able to find the nodes that have their regularity pattern anti-correlated to its own.

This is a common problem of current gossiping services that maintain the same view for two distinct purposes: as a pool of nodes to gossip with and for achieving the goals of the application that is taking advantage of them. To solve such a problem, the nodes would have to maintain different

views for these two purposes. In the specific case of anti-correlation, the view that is to be delivered by the service to the application might be of descriptors of "dead" nodes (not on-line for the moment). This is in contradiction with the fact that the gossiping view must be of "live" nodes that can be gossiped with. This issue cannot be solved by just modifying the way the nodes are sorted and chosen in the same view.

We try to mitigate this issue by considering nodes that we discover during gossiping for all of the timeslots that they are regular in. For example, if we are currently in timeslot t and node p is gossiping with node q which is regular in timeslot t and also in timeslot t+5, then we put node q for consideration in timeslot t+5 also. In this way, even if node p is never on-line during the duration of timeslot t+5, it will still be able to deliver to the application node q as a regular node in t+5 if the application issues a request for that timeslot. This is not a proper solution however, as it relies on the chance that nodes have partly overlapping and partly anti-correlated regularity patterns. This might not always be the case and if node q is never on-line at the same time as p, then they might never get to know of each other at all.

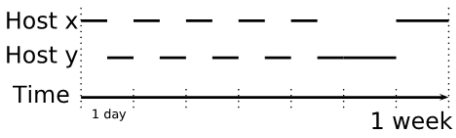


Figure 5.2: An example of anti-correlation as presented in [22]

Chapter 6

Taking Advantage of Node Regularity in PolderCast

The purpose for studying the regularity behavior of nodes and their availability patterns is the idea that we might use this information to our advantage to improve message dissemination atop overlays by increasing hit-ratio. At the same time we would like to maintain other parameters of the dissemination at comparable values with those present when other techniques of increasing hit-ratio are applied like increasing the fanout (number of publication messages each node forwards to other nodes). As we already mentioned in Chapter 1 and 3, there have already been proposals to use such information to improve DHT based distributed storage solutions by trying to reduce the amount of data that is relocated across the nodes under churn. This thesis is focused on P2P topic-based publish-subscribe systems and as such we will try to improve a protocol belonging to this family.

6.1 Introduction

The focus of our research in this chapter is to investigate the feasibility and usefulness of using the regularity concept for improving P2P topic-based publish-subscribe systems. Why focus on this category? First of all, as shown in Section 3.1, various systems designed for message delivery in fully decentralized P2P topic-based publish-subscribe systems cannot achieve 100% delivery (hit ratio) under conditions of churn. Any improvement of this hit ratio is very important. Therefore we have the hypothesis that using the notion of regularity we can improve the hit ratio of P2P topic-based pub-sub systems. Our hopes are that the concept can be used to improve various types of overlays, however, due to the limited scope of this thesis, we focus on improving only one system as a proof of concept.

We seek to use GRID, the service we designed in Chapter 5, for improving the hit ratio of PolderCast [36] for our proof-of-concept purposes. We have created an application that utilizes GRID and that is targeted at improving the hit ratio of PolderCast: *RingsRegular*. We explain its design

later in this chapter.

Why PolderCast? First of all it is a fully decentralized P2P topic-based pub-sub system that possesses the desired properties described in Section 1.2.1. Also, one main reason for this choice is that PolderCast is a modular protocol, composed of 3 layers which benefit from each other's functionality and work in tandem to help make the dissemination better. This modular approach makes such a protocol also easier to extend than other protocols and also makes it possible for us to create a module which does not influence the execution of the original PolderCast, but can be added on top of it in order to improve dissemination. This is important for us to isolate the effects of just the proposed approach of benefiting from node regularity. The modularity makes the integration with the GRID service a lot easier.

PolderCast is shown through the experiments presented in [36] that it responds well to churn, however there is room for improvement, especially if the churn is heavy. That is why we want to try and improve the dissemination by using regularity information about the nodes in the system.

As a basis for the explanations made below are the definitions made in Section 2.3. We explain the extensions that we have done to PolderCast in order to achieve our proof of concept that regularity information could usefully be used in order to improve the hit ratio of P2P topic-based pub-sub systems.

To effectively use the regular property of a subset of the nodes in the network, we should find a way to make them act as a backing bone for the dissemination of messages. Since we have a probabilistic guarantee that these nodes will be on-line for the timeslots where they are regular, we could use these nodes to improve the dissemination in PolderCast in their respective regions of the ring.

6.2 The desired overlay and dissemination

The dissemination boosting module that we have designed is an additional layer of dissemination that is going to work together with PolderCast, and which takes advantage of GRID, the service we described in Chapter 5. By using the same modular approach as PolderCast, for each topic we maintain an additional ring composed only of the regular nodes that are up at the current moment in time of the execution. These per topic rings consisting of regular nodes, will coexist at the same time with the normal rings created in PolderCast. The final desired topology of each of these rings is represented in Figure 6.1. The green links in this image are representative of the deterministic links in the Rings layer of PolderCast, while the blue ones are representative of the deterministic links in the ring that has only regular nodes (denoted by a blue filling in the picture). The dashed lines correspond to the random links that are part of normal and regular rings according to the color.

We can see that the regular nodes are part of both rings. In this

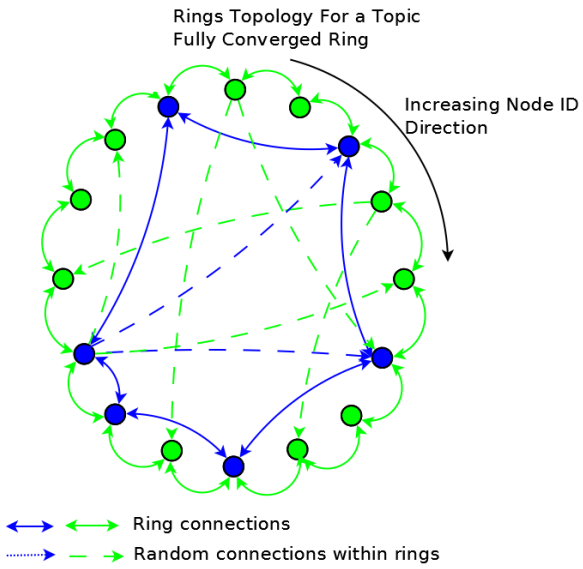


Figure 6.1: The desired topology of the ring for one of the topics once it is converged

way, we hope that as the regular nodes receive a message on the regular ring, they pass it also on in the normal ring, instigating in this way the dissemination in that ring as well. The motivation behind the regular ring is that the dissemination on it is more reliable since these nodes are probabilistically guaranteed to be on-line for the timeslot when the dissemination is happening.

6.3 The protocol structure

The desired topology in Figure 6.1 is a result of the topology created by the Rings layer of PolderCast combined with the one created by the RingsRegular protocol. PolderCast is responsible for the green colored links in the topology, while RingsRegular is responsible for the blue ones. The protocol structure of PolderCast and its inner workings are explained in Section 2.4, therefore we are omitting it here.

RingsRegular is responsible for maintaining rings of regular nodes, for each of the topics a node running it is subscribed to. The population of regular nodes can be different for every timeslot, however dissemination happens only on the timeslot that corresponds to the current point in time, or the *current timeslot*. Therefore the rings that are for the current timeslot are the ones RingsRegular focuses on maintaining and making them converge to the desired topology. Here, each node p tries to find the closest nodes in the ID space for each topic and filters in only the nodes that are regular in the current timeslot.

To fulfill its role, RingsRegular needs to identify the node's own regularity pattern and advertise it when gossiping with other nodes running RingsRegular. Additionally it needs to have a pool of nodes that are regular in the current timeslot and that fulfill its need for topic coverage, in an analogous manner to Rings layer in PolderCast needing

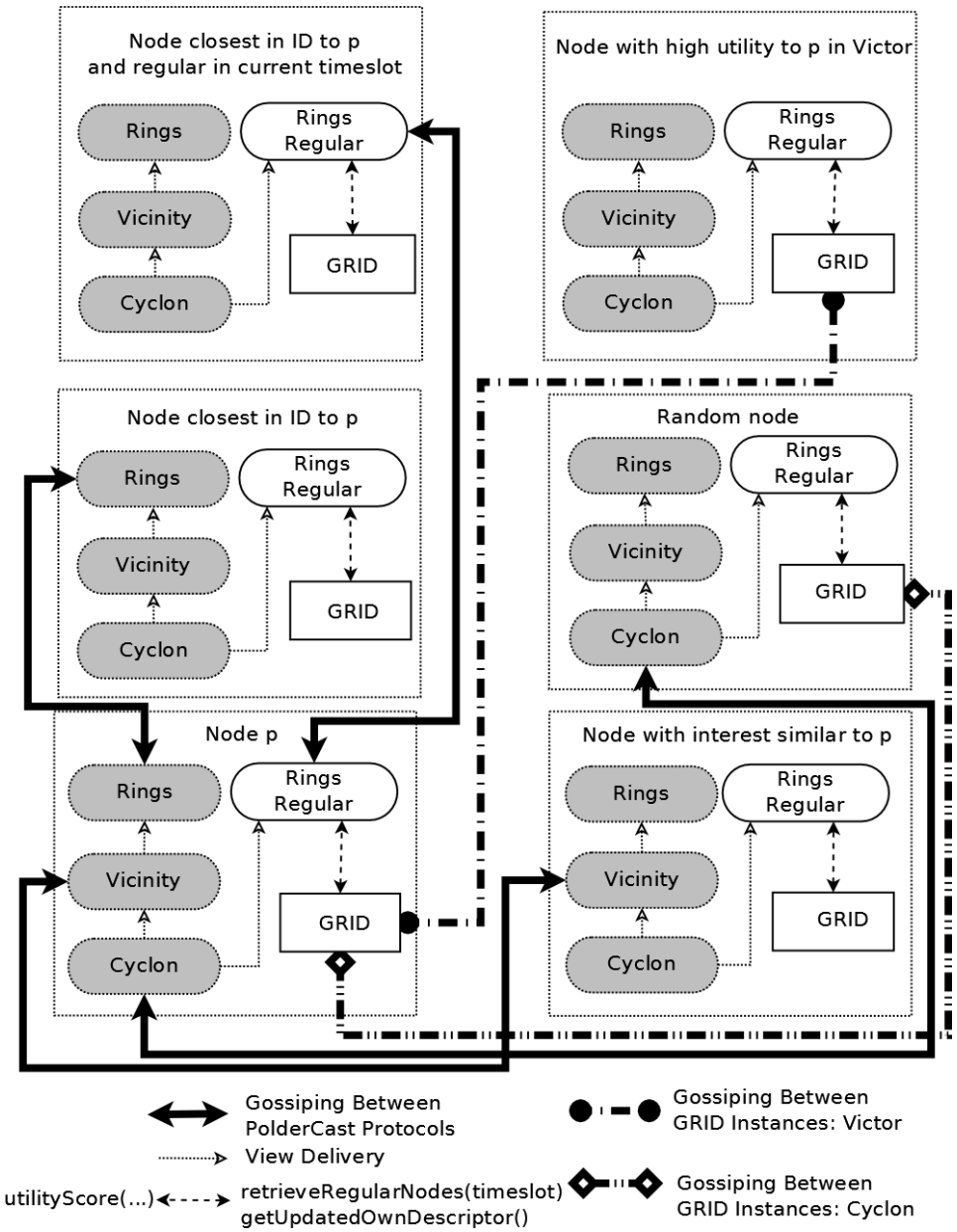


Figure 6.2: The structure of the protocol stack and how the regularity related protocols gossip with each other

Vicinity's view to fulfill its need for topic coverage. For these two reasons, we designed RingsRegular to take benefit of the GRID service we presented in Chapter 5. GRID is able to supply the nodes own regularity patter, while via an implementation of the callback interface of GRID, it can also help RingsRegular converge faster and achieve the desired topic coverage. For the random links, RingsRegular uses regular nodes provided by GRID, or nodes regular in the current timeslot, filtered out from the view of a Cyclon instance. Cyclon is inexpensive [47], therefore we could have run another instance of Cyclon besides GRID and the other protocols. However, since PolderCast is already running an instance of Cyclon, for optimization, we decided to use that one.

The various protocols used in order to achieve the overlay topology in Figure 6.1 is presented in Figure 6.2. In this figure we show in gray the protocols that are already part of PolderCast and in white the additions that we are introducing. All of the protocols in this figure are gossiping ones. This is shown via the various arrows that denote the nodes that any node p in the system would interact with via gossiping at each of the various protocols.

6.4 Algorithms and implementation details

In the following sections we will present the algorithms involved in the implementation of these extensions to PolderCast in more detail. For various notations used in this section, please refer to Table 6.1.

6.4.1 Node Descriptor

The descriptor of the nodes used in this chapter is an extension of the descriptor presented in Data Structure 1. It inherits from it, adding only the information about the various topic subscriptions. The resulting data-structure is presented in Data Structure 3. This is the descriptor that is going to be used in the algorithms presented in this chapter. It should be noted that, in addition to the algorithms presented in pseudocode in this section, every node will query GRID once per timeslot by calling *getUpdatedOwnDescriptor* from its offered interface in order to get the updated descriptor from GRID and obtain the regularity information from it. This regularity information is then integrated into the descriptor used in the RingsRegular protocol.

6.4.2 Use of GRID

Throughout Chapter 5 we explained that GRID offers a callback interface with one function called *utilityScore* which is to be implemented by the application. The *utilityScore* function assigns points to a node q, depending on how much benefit it gives to the reference node p and p's RingsRegular layer. The end formula for this function is presented in Formula 6.1:

Table 6.1: Table of notations used throughout algorithms.

Notation	Meaning
numSeconds-PerTimeslot	The number of seconds assigned to a single timeslot. This represents the timeslot duration. For example if this value is 3600, it means that the timeslot changes every 1 hour. This value is configurable.
totalNumberOf-Timeslots	Represents the total number of timeslots inside a CRB. This is a result of the division of the length of the CRB by *numSecondsPerTimeslot*
currentTimeslot	Refers to the timeslot that corresponds to the current time. The formula for calculating this value is: currentTimeslot $= \frac{\text{currentTimestamp}}{\text{numSecondsPerTimeslot}}$ **mod** totalNumberOfTimeslots
simulationTime	Represents the *currentTimestamp* in the churn trace.
descriptor	Refers to the datastructure the nodes gossip between each other that carries information about them. This datastructure for GRID is presented in 1, while for RingsRegular is the one in 3
V_{Victor}	The view of the Victor protocol, part of GRID. It is composed of one bucket per timeslot. However, since only the bucket corresponding to the current timeslot is active, we usually refer only to $V_{Victor}[currentTimeslot]$
l_{Victor}	Max. capacity in terms of number of descriptors for each bucket in V_{Victor}
g_{Victor}	The gossip length of Victor protocol. It represents the number of descriptors exchanged in a gossip cycle in the Victor protocol.
V_{Cyclon}	It is the view of the Cyclon protocol. This notation is used in GRID, PolderCast and RingsRegular for denoting the view of the respective instance of Cyclon they interact with.
l_{Cyclon}	Max. capacity of the cyclon view used in both GRID and PolderCast.
g_{Cyclon}	Gossip length of Cyclon used in both GRID and PolderCast.
$V_{Vicinity}$	View of Vicinity in PolderCast
$l_{Vicinity}$	Max. capacity of view of Vicinity in PolderCast
$g_{Vicinity}$	Gossip length of Vicinity in PolderCast
V_{Rings}	View of Rings in PolderCast
g_{Rings}	Gossip length of Rings in PolderCast
$V_{RingsRegular}$	The view of the RingsRegular protocol. It is composed of one bucket per timeslot. Each bucket has one set of nodes per topic the node is subscribed to and each of these sets has a max. capacity of k (defined in this table also).
$g_{RingsRegular}$	The gossip length of RingsRegular protocol. It represents the number of descriptors exchanged in a gossip cycle in the RingsRegular protocol.
Reg_x	Regularity pattern of node x. It is a set of timeslots where the node x is regular.
T_x	Set of topics node x is subscribed to.
k	desired coverage for each topic in both Rings and RingsRegular protocols. From the point of view of a node p, k/2 nodes have ID smaller than that of p, while k/2 have ID larger than that of p.

$$score(p,q) = |T_p \cap T_q| \cdot |Reg_p \cap Reg_q| + \sum_{t \in T_p \cap T_q} (|T_p| \cdot |Reg_p|)^{k-count(t)} \quad (6.1)$$

where T_p and T_q represent the set of topics nodes p and q are subscribed to, Reg_p and Reg_q represent the sets of timeslots where p and q are regular respectively; k is the desired coverage factor for each topic for each timeslot and *count(t)* counts the number of nodes for topic t in the current timeslot in the view of RingsRegular layer of node p.

Data Structure 3 Data structure of a *descriptor* of a node, extended for RingsRegular

Descriptor(ID, age, regularityInformation, topicSubscriptions, topicCoverage)
Where:

- *ID* is the node ID
- *Info* is the node's contact information like IP, Port etc.
- *age* a number representing the age of the descriptor, showing how long it has since being communicated with.
- *regularityInformation* is a bit string with one bit per timeslot and the bit is set for the timeslots where the node is regular.
- *topicSubscriptions* is the application specific information; In this case the list of topics the node has interest in.
- *topicCoverage* is the coverage for each of the topics the node is subscribed to in the RingsRegular view. It has a value for each topic, for each timeslot. So topicCoverage[t][T] gives the coverage for topic T on timeslot t. In terms of the view of RingsRegular (Data Structure 4), topicCoverage[t][T] = $|V_{\text{RingsRegular}}[t][T]|$

The idea behind the design of this formula is to favor the second term of the sum when the coverage of some topic in $T_p \cap T_q$ for the current timeslot is $< k$. This is done in order to favor nodes that are subscribed to exactly those topics that are under-covered in the current timeslot by giving them a higher score. Also, when the coverage for all topics in $T_p \cap T_q$ for the current timeslot is achieved, we want the first term of the sum to be dominant in order to try to have nodes in the view that are most similar to the reference node in terms of topics and regularity pattern. There might exist alternative formulae which satisfy the same conditions that we have not explored in this thesis. It would be interesting to explore these alternatives in the future.

The first term of this formula assigns a score proportional to the number of topics and regular timeslots the two nodes have in common. This is done so that, in case this term is to be the dominant one, Victor gets to select those nodes that are most similar in interests and regularity pattern with each other. Such nodes are most probable to be regular in the same timeslots and to be subscribed to the same topics, so that they can still be of benefit to the RingsRegular view in case it is suddenly left without any neighbors in it, resulting from a sudden node departure.

The second term of the formula, on the other hand, gives a score that is exponential to the benefit it brings to the RingsRegular view for current timeslot by covering under-covered topics. This is done so that rings for all the topics node p is subscribed to, achieve k coverage. When the coverage is the lowest, the exponent of the sum is the highest, therefore increasing

the chances node q has the highest score.

As it can be seen $|T_p \cap T_q| \leq |T_p|$ and $|Reg_p \cap Reg_q| \leq |Reg_p|$, therefore $|T_p \cap T_q| \cdot |Reg_p \cap Reg_q| \leq |T_p| \cdot |Reg_p|$, which on the other hand also means that $|T_p \cap T_q| \cdot |Reg_p \cap Reg_q| \leq \sum_{t \in T_p \cap T_q} (|T_p| \cdot |Reg_p|)^{k-count(t)}$ (when $\exists t \in T_p \cap T_q$ for which $count(t) < k$). So the second term of the sum always dominates the result when $count(t) < k$ for any $t \in T_p \cap T_q$.

The first term of the sum, $|T_p \cap T_q| \cdot |Reg_p \cap Reg_q|$, becomes dominant when $\forall t \in T_p \cap T_q \; count(t) = k$, in which case the second term of the sum would practically be $|T_p \cap T_q|$. It is clear that $|T_p \cap T_q| \leq |T_p \cap T_q| \cdot |Reg_p \cap Reg_q|$ for any case when $|Reg_p \cap Reg_q| \geq 1$. We are assured of the fact that $|Reg_p \cap Reg_q| \geq 1$, $\forall q$ because of the way we have designed GRID. As mentioned in Section 5.3 and shown in Figure 5.1, in GRID, prior to *utilityScore* ever being called inside Victor, we apply a Regularity Filter which allows only nodes q that are regular in the current timeslot. Therefore for all timeslots t where p is regular, we are assured that q is regular too, so $|Reg_p \cap Reg_q| \geq 1$. For RingsRegular, p is not very interested in timeslots where it is not regular because p is not part of the regular rings on those timeslots.

Inside GRID, in the Victor layer, after the sorting, the topmost l_{Victor} nodes are kept and the rest are discarded as they have the lowest score and therefore bring the least benefit to the RingsRegular layer for the current timeslot.

Algorithm 7: utilityScore(Descriptor a, Descriptor b)

```
/* Score assignment function used to assign points for the
   sorting process inside the selection function for
   Victor.                                                  */
```

Input: Descriptor a - descriptor of the node used as a reference
Input: Descriptor b - descriptor of the node that is to be measured for the utility it brings to the reference Descriptor a
Output: score - a number which quantifies the utility of *b* towards *a*, the higher, the more preferred node *b* is for node *a*

```
1  regularityA ← a.regularityInformation;
2  regularityB ← b.regularityInformation;
3  numTimeslotsInCommon ← |reguarityA∩ regularityB|;
4  topicsA ← a.topicSubscriptions;
5  topicsB ← b.topicSubscriptions;
6  topicIntersection ← topicsA∩ topicsB;
7  utilityScore ← |topicIntersection|· numTimeslotsInCommon;
8  forall the topic ∈ topicIntersection do
9      count ← a.topicCoverage[currentTimeslot][topic];
10     utilityScore + = (|topicsA| · |regularityA|)^(k−count);
       /* k is the desired topic neighbor coverage in the
          RingsRegular layer                                */
11 return utilityScore;
```

Some of the information used inside Algorithm 7 is clearly extraneous to GRID. However, this is ok since the signature of the function is adhering to the interface that GRID is expecting, while the implementation is done outside of GRID. GRID itself uses the implementation of *utilityScore* as a black-box. Therefore all of the information about topic subscriptions of the nodes is available to the application.

6.4.3 RingsRegular

RingsRegular inherits a lot of its behavior from the Rings protocol used in PolderCast. It is a gossiping protocol. Its goal is to try to form a ring of the nodes that are regular in the current timeslot, for each of the topics a node p is subscribed to. Similarly to the way the Rings layer in PolderCast works, in the RingsRegular layer any node p tries to have k neighbors coverage for each of the topics for the current timeslot and ideally these neighbors should be as close to node p's ID in the ID space as possible in order for the ring to converge. This means that $k/2$ neighbors should be with lower ID than p and $k/2$ with a higher ID.

The population of regular nodes is timeslot specific, which means that for each timeslot there is a potentially different set of nodes that is regular in it. What happens when the time of the current timeslot has passed? This means that the next timeslot becomes the current one and the set of regular nodes is different. Does this mean that RingsRegular has to start building the rings from scratch?

As we explained in Section 2.3, timeslots repeat themselves every CRB. We also know that the regularity of a node does not change often and several passes of the CRB are needed for it to change. This means that for RingsRegular we could use the same technique we used in Victor (Section 5.5.3) and we make RingsRegular maintain a view that has one *bucket* per timeslot. Every bucket is a view in itself that contains the same information that the view of the Rings layer in PolderCast contains. The difference is that in each bucket we put only nodes that are regular in the timeslot that corresponds to this bucket. In this way we can keep information about the neighbors the RingsRegular protocol has had for any timeslot. In this way, when the time of the same timeslot t comes again the next pass of the CRB, RingsRegular running on a node p will have k nodes for each topic p is subscribed to, that are regular in t and that are closest in ID to p. There is a chance that the regularity pattern of these neighbors might have changed since the last pass of the CRB, however this is not the general case. As we mentioned previously, it takes several passes of the CRB for the regularity pattern of nodes to change.

The data structure for the view of RingsRegular is represented in Data Structure 4 and we denote it by $V_{\text{RingsRegular}}$. Depending on the number of topics and on the chosen length of a timeslot, this data structure can have a considerable footprint in memory (for 1000 topics, 168 timeslots, 4 neighbors per each topic-timeslot pair, and 50 bytes per descriptor would mean 32 MB of data in memory). For a single node, nowadays this is not much and could be considered totally acceptable. It might make sense to

Data Structure 4 The structure of the view of the RingsRegular protocol, $V_{RingsRegular}$.

Descriptor[*totalNumberOfTimeslots*][$|T_p|$][k] $V_{RingsRegular}$
Where:

- If we denote by p the local node running RingsRegular, then T_p is the set of topics p is subscribed to.
- There is one bucket per timeslot. Each of these buckets has one set of max. capacity of k node descriptors per topic $T \in T_p$. So each bucket is a set of sets of descriptors.
- To access the bucket of a specific timeslot t, we denote $V_{\text{RingsRegular}}[t]$.
- To access the set of nodes in the view for a specific timeslot t and topic T, we denote $V_{\text{RingsRegular}}[t][T]$. Maximal capacity of $V_{\text{RingsRegular}}[t][T]$ is k.

save the information on disk and retrieve it when it is required to be used.

Let us consider RingsRegular running on a node p. Every gossiping cycle, p initiates a gossip request with another node q from its view $V_{\text{RingsRegular}}$[currentTimeslot]. The algorithm for the gossip initiation method is presented in Algorithm 8. At the beginning of this algorithm, node p gathers a set V of the nodes in all of the protocols RingsRegular sits on top of, namely Cyclon and the view of Victor for the current timeslot from GRID. From this set, p filters only the nodes that are regular in the current timeslot and updates its own view by calling the method considerNodes(V), the algorithm of which is presented in Algorithm 10.

In considerNodes(V), node p tries to improve its view for the current timeslot, as well as for all the timeslots where the nodes in V are regular. For each node n in V, for all the timeslots t_n where n is regular and for all the topics T both p and n are subscribed to, p executes the method applyHashFunction(n, $V_{\text{RingsRegular}}[t_n][T]$, p). The algorithm for the applyHashFunction method is presented in Algorithm 11. This method determines if any of the neighbors in $V_{\text{RingsRegular}}[t_n][T]$ should be replaced by n based on n's ID proximity to p's ID. The decision is made keeping in mind that $k/2$ neighbors should be with lower ID than p and $k/2$ with a higher ID than p. For clarity we have provided also the algorithms for two other methods used inside of the applyHashFunctionMethod: *getClockwiseDistance* (Algorithm 15) which gives the distance in ID between two descriptors and *compareDistance* (Algorithm 16) which is used in sorting nodes according to their ID distance to a reference node.

After returning from *considerNodes*, p chooses a random topic t on which which to focus the gossiping. After this, node p increases the age of all neighbors in $V_{\text{RingsRegular}}$[currentTimeslot][t] and chooses its gossiping partner q, the node with the largest age in $V_{\text{RingsRegular}}$[currentTimeslot][t]. The oldest node is chosen so that we gossip with the least recently used

neighbor because we want to contact all our neighbors periodically in a consistent manner so that we can remove dead neighbors, get fresh copies of their descriptors and in general for churn handling. The least recently used fashion of choosing the gossiping partner is also used in Cyclon, Vicinity, Rings etc. for the same reasons.

After selecting its gossiping partner q, node p selects the best neighbors to send to q via the method selectBestNeighborsToSend(q, t), the algorithm of which is presented in Algorithm 12. This method tries to find $g_{RingsRegular}$ nodes in total. First it chooses k nodes from $V_{\text{RingsRegular}}[\text{currentTimeslot}][t]$ with ID closest to q, k/2 nodes with ID lower and k/2 ones with ID higher than q's ID. If $g_{RingsRegular} > k$ then to fill the other empty places it keeps adding k nodes for each of the topics p and q are both subscribed to, in the same fashion as for topic t, until it has selected $g_{RingsRegular}$ nodes in total. The list of node descriptors returned by selectBestNeighborsToSend(q, t) is sent to q in the form of a gossip request.

We do not know if node q is on-line or if it is off-line. Normally we should remove q from $V_{\text{RingsRegular}}[\text{currentTimeslot}]$ and add it back in case it replies to our gossip request. At least this is what is done in Cyclon, Vicinity etc. However, removing node q from $V_{\text{RingsRegular}}[\text{currentTimeslot}]$ might have a negative effect on the dissemination of messages in the rings where q was a neighbor of p, until q replies back if it does, or p finds a better ring neighbor. To avoid this, we do not remove q immediately after sending it a gossip request. Instead, we set a variable *pendForRemoval* to the value of the descriptor of q. The next time node p executes Algorithm 8 (next gossiping cycle), if *pendForRemoval* still holds the value of q, then q is removed from $V_{\text{RingsRegular}}[\text{currentTimeslot}]$. If *pendForRemoval* is not set to anything, it means that q has replied to our gossip request. This can be seen in lines 5-9 of Algorithm 8. The method executed when q replies to the gossip request is presented in Algorithm 13 and here is where *pendForRemoval* is unset.

We explained what happens when node p initiates a gossip request and when it gets a response. For completeness we have also shown what node p does when it receives a gossip request in Algorithm 14. The steps followed in this algorithm are very similar to those executed when p is the one initiating a gossip request, therefore they do not need further explanation.

6.4.4 Dissemination Algorithm

A dissemination scenario where the rings are converged for a specific topic is shown in Figure 6.3. Let's suppose that dissemination starts at a regular node 'p'. This node will try to disseminate the message in both the regular ring (marked in blue in Figure 6.1) and the normal ring (marked in green in Figure 6.1). For both of the rings the node that starts the dissemination sends the message to its predecessor, successor and $f - 1$ other random neighbors across the respective ring, where f is the fanout parameter.

When a node gets a message, it tries to forward it to its successor and predecessor in the normal ring and, if the node is regular in the current timeslot, it tries to forward it to its successor and predecessor in the regular

ring too, unless it received the message from any of these nodes. Since the node has to provide a fanout f for each of the rings, it forwards the remaining number of messages to random nodes across the normal and regular ring. This can be seen in step two and three of the dissemination in Figure 6.3.

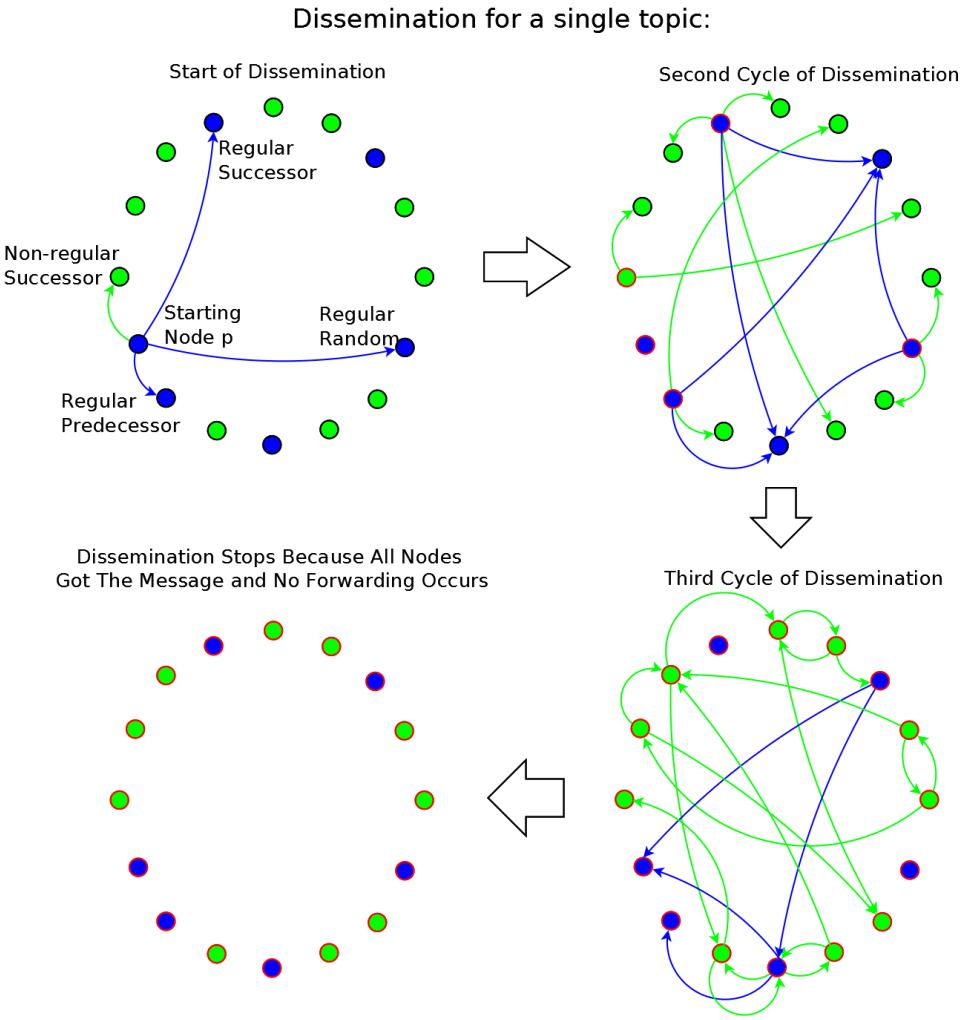


Figure 6.3: A dissemination scenario for one topic

Algorithm 8: Next cycle for RingsRegular protocol that handles the gossiping cycle of the RingRegular layer.

Data: pendForRemoval - a variable used to hold either the descriptor of a node, or NULL(nothing). If holding a descriptor, it means that the node corresponding to that descriptor has been gossiped with and has not replied back. If NULL, it means that the node we gossiped with has already replied. The value of pendForRemoval is also modified in Algorithm 13

1 currentTimeslot $\leftarrow \frac{simulationTime}{numSecondsPerTimeslot}$ **mod** totalNumberOfTimeslots;
2 $V_{Victor} \leftarrow$ GRID.retrieveRegularNodes(currentTimeslot);
/* Get fresh descriptor of the local node (called node 'p' for simplicity) from GRID. It contains updated *regularityInformation*, age=0. */
3 $p \leftarrow$ GRID.getUpdatedOwnDescriptor();
/* Update *p* with latest *topicCoverage* information as described in Data Structure 3. Implementation shown in Algorithm 9 */
4 p.topicCoverage $\leftarrow$ getUpdatedTopicCoverage();
/* If pendForRemoval is set to hold the value of a descriptor d, it means that a gossip request has been sent to the node represented by d and it has not replied since. Therefore we presume d is off-line and remove d from $V_{\text{RingsRegular}}$[currentTimeslot] for all the topics where d appears. */
5 **if** $pendForRemoval! = NULL$ **then**
6 **forall the** $topic \in p.topicSubscriptions$ **do**
7 **if** $pendForRemoval \in V_{RingsRegular}[currentTimeslot][topic]$ **then**
8 Remove pendForRemoval from $V_{\text{RingsRegular}}$[currentTimeslot][topic];
9 pendForRemoval $\leftarrow$ NULL;
10 V $\leftarrow \{x | x \in \{V_{Cyclon} \cup V_{Victor} \cup V_{RingsRegular}[\text{currentTimeslot}] \cup \{p\}\} \wedge$ x.regularityInformation[currentTimeslot] = 1 $\}$
/* Update own view using the collected neighbors from all protocols. In this step the hash function used to calculate IDs and distances is used and the predecessor and successor neighbors for each topic are updated if better ones are found */
11 considerNodes(V);
12 t $\leftarrow$ Select a topic to gossip about in a random fashion;
13 Increase age of all descriptor$\in V_{\text{RingsRegular}}$[currentTimeslot][$t$] by 1;
14 q $\leftarrow$ Select descriptor with oldest age in $V_{\text{RingsRegular}}$[currentTimeslot][t];
15 q.age $\leftarrow$ 0;
/* Pend q for removal from $V_{RingsRegular}[currentTimeslot][t]$ if it does not reply. */
16 pendForRemoval $\leftarrow$ q;
17 neighborsToSend $\leftarrow$ selectBestNeighborsToSend(peerToGossip, t);
18 Transmit *neighborsToSend* to q;

Algorithm 9: int[][] getUpdatedTopicCoverage()

Data: T_p - the set of topics the local node is subscribed to.
Output: A two dimensional array of integers *topicCoverage* so that topicCoverage[t][T] = $|V_{\text{RingsRegular}}[t][T]|$ for all timeslots t and for all topics T the local node is subscribed to.

```
1 int[][] topicCoverage;
2 for t=0 to totalNumberOfTimeslots do
3     forall the T ∈ T_p do
4         topicCoverage[t][T] = |V_RingsRegular[t][T]|;
5 return topicCoverage;
```

Algorithm 10: void considerNodes(Descriptor[] nodeList)

```
/* This method tries to improve V_RingsRegular by considering a
   list of nodes                                                 */
```

Input: Descriptor[] nodeList - list of node to be considered for improving own view

```
/* Get fresh descriptor of the local node (called node 'p'
   for simplicity) from GRID. It contains updated
   regularityInformation, age=0.                                 */
1  p ← GRID.getUpdatedOwnDescriptor();
/* Update p with latest topicCoverage information as
   described in Data Structure 3.  Implementation shown in
   Algorithm 9                                                   */
2  p.topicCoverage ← getUpdatedTopicCoverage();
3  topicsSelf ← p.topicSubscriptions;
4  foreach node ∈ nodeList do
5      topicsNode ← node.topicSubscriptions;
6      foreach timeslot ∈ {t|node.regularityInformation[t] = 1} do
7          foreach topic ∈ topicsSelf ∩ topicsNode do
8              if |V_RingsRegular[timeslot][topic]|<K then
9                  V_RingsRegular[timeslot][topic] ←
                   V_RingsRegular[timeslot][topic] ∪ {node};
10             else
11                 applyHashFunction(node, V_RingsRegular[timeslot][topic],
                   p);
```

Algorithm 11: applyHashFunction(Descriptor candidate, Descriptor[] neighbors, Descriptor selfDescriptor)

Input: Descriptor candidate - the candidate node to be considered for the ring
Input: Descriptor[] neighbors - current neighbors for the corresponding ring.
Input: Descriptor selfDescriptor - descriptor of the node running the algorithm
Result: Considers if any of the descriptors in *neighbors* should be replaced by *candidate* based on its ID distance from *selfDescriptor*

```
1  candidateDistance ← getClockwiseDistance(selfDescriptor,
   candidate);
2  sort(selfDescriptor, neighbors) using compareDistance(...) function;
   /* The rightmost and leftmost descriptors in neighbors now
      represent the nodes closest and furthest in ID to
      selfDescriptor.  So these descriptors could be
      appropriate successors and predecessors to the local
      node.  Middle-list descriptors are discarded if needed,
      as they are less useful.                                  */
3  worstSuccessor ← neighbors[⌈|neighbors|/2⌉ − 1];
4  worstPredecessor ← neighbors[⌈|neighbors|/2⌉];
5  worstPredecessorDistance ← getClockwiseDistance(selfDescriptor,
   worstPredecessor);
6  worstSuccessorDistance ← getClockwiseDistance(selfDescriptor,
   worstSuccessor);
7  if candidate.ID != worstSuccessor.ID and candidateDistance <
   worstSuccessorDist then
8      neighbors.remove(worstSuccessor);
9      neighbors.add(candidate);
10 else
11     if candidate.ID != worstPredecessor and
       candidateDistance>worstPredecessorDist then
12         neighbors.remove(worstPredecessor);
13         neighbors.add(candidate);
```

Algorithm 12: Descriptor[] selectBestNeighborsToSend(Descriptor d, Topic t)

Input: Descriptor d - the node for which the neighbors are to be selected
Input: Topic t - the topic for the ring of which we are gossiping
Output: lst - list of neighbors from own view closest in ID to d
Result: Selects the best neighbors to send to d for the topic t

1 $V_{Victor} \leftarrow$ GRID.retrieveRegularNodes(currentTimeslot);
2 lst $\leftarrow \{x | x \in \{V_{Cyclon} \cup V_{Victor} \cup V_{\text{RingsRegular}}[\text{currentTimeslot}][\text{t}]\} \wedge$ x.regularityInformation[currentTimeslot] = 1 $\wedge t \in$x.topicSubscriptions$\}$
/* Do not send d to the node represented by d itself. */
3 **if** $d \in lst$ **then**
4 Remove d from lst;
/* Send at most k descriptors with closest ID to d for topic t */
5 **if** $|lst| > k$ **then**
6 sort(lst) using compareDistance(...) with d as a referenceNode;
/* The rightmost and leftmost descriptors in lst now represent the nodes closest and furthest in ID to d. So these descriptors could be appropriate successors and predecessors to d. Middle-list descriptors are discarded as less useful. */
7 **while** $|lst| > k$ **do**
8 Remove lst[$\frac{|lst|}{2}$] from lst;
/* While $|lst| <= g_{RingsRegular}$, we perform the same steps as we did for topic t, for other random topics both the local node and d are subscribed to. The goal is to get $g_{RingsRegular}$ descriptors in lst that are closest in ID to d */
9 **while** $|lst| <= g_{RingsRegular}$ **do**
10 topic $\leftarrow$ Select random topic from d.topicSubscriptions;
11 $N \leftarrow \{x | x \in \{V_{Cyclon} \cup V_{Victor} \cup V_{\text{RingsRegular}}[\text{currentTimeslot}][\text{topic}]\} \wedge$x.regularityInformation[currentTimeslot] = 1 $\wedge t \in$x.topicSubscriptions $\}$
/* Perform steps similar to the ones from 3 to 8 in this algorithm. */
12 **if** $d \in N$ **then**
13 Remove d from N;
14 **if** $|N| > k$ **then**
15 sort(N) using compareDistance(...) with d as a referenceNode;
16 **while** $|N| > k$ **do**
17 Remove $N[\frac{|lst|}{2}]$ from N;
18 **forall the** *descriptor* $\in N$ **do**
19 **if** *descriptor* $\notin lst$ ***and*** $|lst| < g_{RingsRegular}$ **then**
20 lst.add(descriptor);
21 **return** lst;

Algorithm 13: Algorithm what to do when receiving a gossip response

Input: q - the node that is replying to a gossip request
Input: V_q - Received neighbors from q
Data: pendForRemoval - Same variable used also in Algorithm 8

```
1 if pendForRemoval=q then
      /* The node q replied back, so no need to remove it.
         So we unset variable pendForRemoval                    */
2     pendForRemoval ← NULL;
3 considerNodes(V_q);
```

Algorithm 14: replyToGossipRequest(Descriptor q, Descriptor-Set received)

```
1  V_Victor ← GRID.retrieveRegularNodes(currentTimeslot);
   /* Get fresh descriptor of the local node (called node 'p'
      for simplicity) from GRID. It contains updated
      regularityInformation, age=0.                             */
2  p ← GRID.getUpdatedOwnDescriptor();
3  Update p with latest topicCoverage information as described in Data
   Structure 3;
4  V ← {x|x ∈ {V_Cyclon ∪ V_Victor ∪ V_RingsRegular[currentTimeslot] ∪
   {p}}∧x.regularityInformation[currentTimeslot] = 1 }
5  t ← Select a topic to gossip about in a random fashion;
6  neighborsToSend ← selectBestNeighborsToSend(peerToGossip, t);
7  Transmit neighborsToSend to q;
8  q.age ← 0;
9  V ← V∪ received ∪{q};
   /* Update own view using the collected neighbors from all
      protocols.                                                */
10 considerNodes(V);
```

Algorithm 15: long getClockwiseDistance(Descriptor p, Descriptor q)

Input: Descriptor p, Descriptor q - the two descriptors to be compared based on their ID, how distant they are in the ring
Output: returns the clockwise distance between the two nodes starting from p

```
1 if p.ID <= q.ID then
2     return q.ID − p.ID;
3 else
4     return MAX_VALUE − p.ID + q.ID;
      /* MAX_VALUE represents the total ID space, the total
         amount of nodes supported by the system, usually 2^128
         or 2^256                                               */
```

Algorithm 16: int compareDistance(Descriptor referenceNode, Descriptor a, Descriptor b)

Input: Descriptor referenceNode - the descriptor of the node that is taken as a base to measure the distance of the other two inputs
Input: Descriptor a, Descriptor b - the descriptors to be measured the distance with regards to node *referenceNode*
Output: determines which of the nodes *a* or *b* is closer to the *referenceNode* with regards to distance. It returns 0 when *a* and *b* are equally distant from *referenceNode*, +1 when node *a* is preferred and -1 when node *b* is preferred.

```
1 distA ← getClockwiseDistance(referenceNode, a);
2 distB ← getClockwiseDistance(referenceNode, b);
3 if distA = distB then
4     return 0;
5 else
6     if distA > distB then
          /* Prefer a                                    */
7         return +1;
8     else
          /* Prefer b                                    */
9         return -1;
```

6.5 Possible challenge: Dealing with not synchronized clocks

We mentioned in Section 4.4.3 that there could be a problem resulting from the fact that clocks in distributed systems are not synchronized. Machines rely on their local clocks and do not share a global clock, therefore synchronization is never fully achievable. The correct functioning of GRID and RingsRegular is based on the idea that nodes are able to determine the timeslot they are currently in. If the clocks of different nodes are out of sync, we could have a situation where different nodes think that they are in different timeslots of regularity. This could be problematic for the period prior to the end of one timeslot and after the beginning of the next timeslot. This problematic period of time has the duration of double the maximum amount of time the nodes are out of sync with each other.

Let us denote the maximum amount of time that clocks can be allowed to drift out of sync as α, any two adjoining timeslots t and $t+1$ and θ the time when t ends and t+1 starts. The solution to this problem would be to allow nodes use as ring neighbors in RingsRegular regular nodes from both t and $t+1$ for the period of time $[\theta - \alpha, \theta + \alpha]$, under the assumption that α is far shorter than the length of a timeslot. In this way, for a brief moment of time, some nodes might be considered regular while they are not. If any of such nodes are actually dead (not on-line), they will be removed from the view via the process of gossiping. Gossiping in a real world P2P system should happen several times a minute. As a result dead nodes should be removed quickly from the view. Therefore by the end of this interval of disruption, the set of nodes that are regular in $t+1$ will be consolidated.

However, based on the results in Section 4.4.3, the percentage of nodes that actually change their regularity status from *regular* to *not regular* from one timeslot to another is very small ($\leq 11\%$). So even if we do not take the clock drift problem into consideration, it would not affect the execution of the protocols in any considerable way.

6.6 Experiments' setup

6.6.1 PeerSim

PolderCast was implemented in Java and for the P2P simulator PeerSim [30]. PeerSim is a discrete event simulator developed in Java. It is not multi-threaded and not distributed, therefore it has limited scalability with regards to the number of nodes and amount of memory and processing needed for each node. It provides facilities for simulating cyclic and event driven protocols. The first type of protocols are those that are executed every time a certain amount of time passes, while the later are protocols that react to events like messages from other peers.

While PeerSim is easy to use, highly configurable and extensively used in the research community ([21, 47, 48, 50] etc.)[1], it also suffers from some

[1] http://peersim.sourceforge.net/pubs/desc.html

issues. Although it claims to scale well even to large numbers of nodes, in practical setups, it is not very suitable for large populations of nodes. This is mainly due to the fact that the simulations would then take a long time to execute and the memory requirements would be quite high. While Java is a very nice language to program with and allows programmers to focus on the functionality they want to achieve without worrying about the memory management or other smaller details, it is less suitable for memory-intensive simulations, contributing to a bigger memory footprint, less efficient memory management and an overall slower execution. For example, some of the simulations we ran take around 48 hours to complete and use up to 20-30GB of memory for 1000 nodes in total in the system, even with just PolderCast.

6.6.2 Datasets and Availability Traces Used

Subscriptions dataset

For testing our system we needed a trace of subscriptions in a system in order to represent links between the nodes. For this purpose we used a Twitter dataset produced and studied by [24]. This dataset contains information about 41.7 million users and their *following* and *being followed* relations. These relations are unidirectional. This means that if one user is following another user, it is not mandatory that the relationship is symmetric. The *following* relationship translates to the *subscribed to* relationship in publish-subscribe systems.

For our experiments we chose a dataset of 1000 nodes from Twitter used previously in [36]. The method of getting this dataset is explained in [36] and is based on a method used in [33, 34], therefore we are not going into details about reasoning behind this method. In short, the procedure of getting a subset of the dataset is: we start from a random sample of initial nodes; the social graph is traversed using breadth first search until the desired number of nodes is reached and all edges between them are included in the subset. The resulting number of topics per node and number of nodes per topic is plotted in Figure 6.4. In Twitter jargon they represent the followees and the followers respectively.

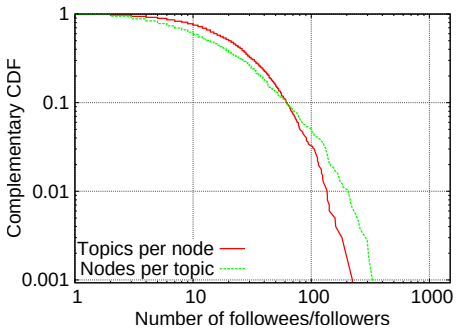


Figure 6.4: CCDF of number of topics per node and number of nodes per topic in the Twitter subset of 1000 nodes we are using

Availability Trace

As the availability trace we decided to use the Skype one which was discussed earlier in Section 4.2.2. There were multiple reasons for this choice. The Skype trace has a good ratio of nodes that are on-line at any point in time to the total number of nodes in the trace. In the KAD trace this ratio is much lower, as we explained in Chapter 4. This means that if we use the KAD trace, in order to have the same amount of nodes on-line at any time as during experiments with the Skype trace, we would have to run experiments with a higher number of total nodes in the system. This fact makes the Skype trace more appropriate in our case since we explained the problems with scalability of our simulator PeerSim. Running higher numbers of overall node population in the experiments would mean that either we would not have enough RAM to run the experiments at all, or the time required to complete the experiments would be too long.

As discussed in Chapter 4, the Skype trace offers also a higher ratio of regular nodes to the number of nodes that are on-line at any time, compared to the KAD trace. If this ratio is low, there might not be enough regular nodes on-line throughout the experiments to aid in message dissemination, especially since we cannot run experiments with large numbers of nodes. This situation could lead us to underestimate the improvement in hit-ratio we get from using the approach discussed in this chapter. Our approach works best when there are many regular nodes in proportion to all the nodes that are on-line throughout the experiments.

For these reasons we choose to use the Skype trace as the availability trace for our experiments. The results of these experiments however apply to all systems that have a similar ratio of regular nodes to total number of on-line nodes at any given point in time. Unfortunately we could not run the same experiments with the KAD trace in order to see the results in a system where this ratio is lower. This is left as a future work, when the scalability issues with our PeerSim implementation are overcome.

Latency

As done also in [36], we are also using *King dataset* [19] for modeling communication latency between the nodes.

6.6.3 Parameters Used

Since we wanted to focus on improvements in hit ratio compared to PolderCast, we decided to use the same parameters with regards to view size and gossip length for the protocols. These parameters were also the ones used to benchmark PolderCast in [36]. Therefore the parameters chosen for the simulations were:

- $l_{Cyclon} = 20$ (Cyclon view length),
- $g_{Cyclon} = 10$ (Cyclon gossip length),
- $l_{Vicinity} = l_{Victor} = 20$ (Vicinity and Victor view length),

- $g_{Vicinity} = g_{Victor} = 10$ (Vicinity and Victor gossip length),
- $k = 4$ (desired coverage in Rings and RingsRegular layer),
- $g_{Rings} = g_{RingsRegular} = 10$ (Rings and RingsRegular gossip length)

For simulation purposes we have used only one instance of Cyclon due to the fact that Cyclon is used for the same purpose in PolderCast and in GRID. The view of Cyclon in the simulations is therefore consumed by both PolderCast and GRID as it would have been if there were two separate instances of it.

In particular, we test sensitivity to the speed at which the churn is played, which translates to seconds elapsed between consecutive cycles of gossiping and dissemination. It is at higher speeds of playback of the churn trace that we expect better improvements to the hit ratio. Therefore we decided to run experiments with the following churn speeds:

- Slow churn speed with gossip and dissemination cycles every 100 seconds.
- Medium churn speed with gossip and dissemination cycles every 200 seconds
- High churn speed with gossip and dissemination cycles every 1000 seconds.

6.6.4 Metrics to be observed

We described the main metrics that we are going to observe in the new protocol in Section 2.5. Our main focus was on improving the hit ratio of PolderCast with the use of regularity information, so hit ratio will be the main parameter that we focus on in our analysis. However, improvements in hit ratio that lead to great deterioration in other aspects would not be helpful. That is why we are also measuring other key parameters of the overlay, presented also previously in Section 2.5. Here is how we measure them:

Hit Ratio: Each cycle we disseminate a message for all the topics in the system by selecting a random node for each of them as a starting point. The hit-ratio we present in the results is the average of the hit-ratio of all the topics. The settings of our simulations make it so that the dissemination process is finished before we measure the hit-ratio. From the calculations are excluded the nodes that have executed less than 10 cycles of gossiping in the Rings and RingsRegular combined. This is done in order to allow a node that just came into the system to achieve connectivity for the topics it is subscribed to.

Node degree: This metric is calculated as the size of the set of unique nodes that a node has in the views of each protocol it runs. If we are measuring the degree of node p, then the in-degree of p is the number of nodes that have p in their views. The out-degree is the number of nodes

that node p has in its views. The total degree is the union of the sets of nodes that have p in their views with the set of nodes that are in p's views.

Number of messages sent and received: we measure this metric by putting a counter for received messages and one for sent messages on all protocols, especially the dissemination protocol which we are more interested in. The bulk of the messages sent and received is from the dissemination protocol.

Path length: we measure this parameter by increasing the age of a message each time it is forwarded to another node and recording the age of the message when it is received for the first time by a node.

Message redundancy: we put a counter on each node for each message it receives. This counter starts with 1 and then every time that same message is received, the counter is increased therefore counting how many times that message has been received. Therefore, a redundancy of 1 means that the message has been received exactly once. Any increase to this number identifies duplication and therefore overhead.

We could have also measured the bandwidth consumption of our new approach compared to PolderCast, however this parameter is easily calculated. The overlay maintenance messages are the ones that are transmitted by Cyclon, Vicinity, Rings, Victor and RingsRegular protocols. Out of these, only Victor and RingsRegular are added on top of the protocols that are part of PolderCast. The frequency with which these protocols communicate is the same for all of them. Therefore in the regularity approach we have only $\frac{2}{3}$ more messages compared to PolderCast. Each of the messages would be bigger in bandwidth by just the regularity information which is expressed in one bit per timeslot, which means 24 bits for the daily CRB and 168 bits for the weekly one. As for the topic messages dissemination, we do not know the size of the messages that could happen in real life, therefore we take as a reference only the number of messages sent and received.

6.6.5 Baseline and simulations with regularity

Our approach is based on the concept of providing a backup dissemination of the message in the regular ring where we presume its dissemination is safer due to a lower probability of a disconnection in this ring. We explain how this is achieved throughout Chapter 6. The benchmark for which PolderCast was tested was with a fanout equal to 2, and that is what we are maintaining in the regularity approach also. Adding to what we mentioned earlier, this means that the regular nodes will effectively have a fanout of 4 because of the dissemination on two parallel rings. In order to be fair in our comparisons and in order to see if the benefits in hit ratio come from the increased fanout in some of the nodes or from the regularity approach, we decided to compare it with PolderCast with fanout $F = 4$ besides the one with $F = 2$.

Based on the results of the analysis done in Chapter 4 we decided to run our experiments for the regularity thresholds $\gamma = 70\%$ and $\gamma = 75\%$ since these seem to be the values where the population of regular nodes with weekly period changes significantly. Also based on these results we

decided to use 3600 seconds as timeslot length since it was concluded that it does not really matter as this choice does not affect the number of regular nodes or the accuracy of the prediction. This choice also helps us lower the number of timeslots in total for the experiments and therefore lower the memory and processing requirements for the machine running the simulations.

All simulations are dependent on a seed of the random number generator for their execution. Therefore we ran each of the experiments 4 times, with different seeds and then took as final results the averages of the values that came out of these four runs.

The simulations were executed on nodes from the Abel computing cluster of the University of Oslo, which is made up of nodes that have 64GB of RAM and 16 CPU cores. This was done due to the time and memory requirements of the simulations.

6.7 Experimental results

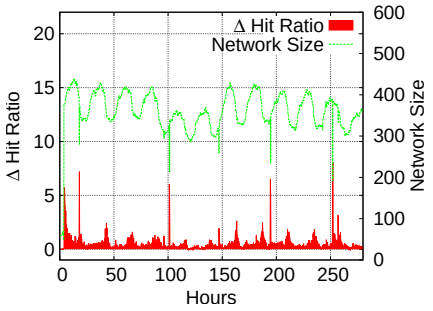
In this section we explain the results of the simulations we ran. The discussion will be with regards to the parameters mentioned in the Section 6.6.4.

6.7.1 Hit Ratio

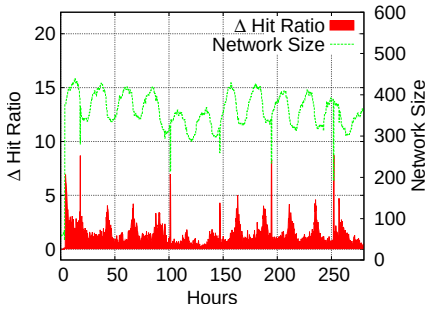
In order to see the improvements achieved by using regularity against PolderCast we are presenting the graphs that show the difference in hit ratio between the former and the later. These are presented in Figures 6.9, 6.8, 6.7, 6.6 where the baseline is PolderCast with $F = 4$. In order to see the full picture of the benefits in this aspect, we decided to compare the results between PolderCast runs with $F = 2$ and $F = 4$. These are presented in Figure 6.5a, 6.5b and 6.5c. In these images we see the difference in hit ratio between the run with $F = 4$ vs. the one with $F = 2$. This gives us a method of comparing the benefits of using the regularity approach versus simply increasing the fanout of the dissemination protocol in the normal PolderCast.

As can be seen in Figure 6.5a, 6.5b and 6.5c, Poldercast with $F = 4$ has a lot of improvement in hit-ratio compared to PolderCast with $F = 2$. This is due to the increased fanout of nodes, which makes the entire dissemination more churn resilient. In Figure 6.6, 6.7, 6.8 and 6.9 we can see that the regularity approach outperforms even PolderCast with $F = 4$, which means that the increase in hit ratio is not only due to the increased fanout of the nodes, but also due to the regular ring acting as a safe dissemination ring for the message. These nodes also initiate dissemination on the normal ring, therefore improving dissemination in that ring also.

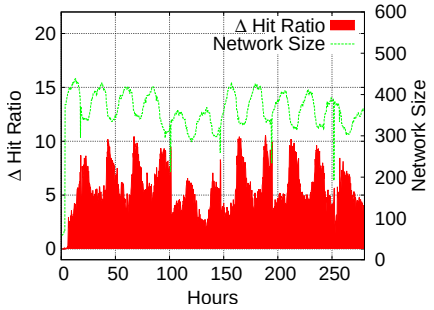
Another thing that can be clearly seen is that the biggest improvement coming from the regularity approach happens at higher churn speeds. At the same time this means that using the regularity approach, overlay maintenance messages can be exchanged in a less frequent manner as that



(a) Hit-ratio improvement when using PolderCast with F=4 versus PolderCast with F=2, slow churn speed



(b) Hit-ratio improvement when using PolderCast with F=4 versus PolderCast with F=2, medium churn speed



(c) Hit-ratio improvement when using PolderCast with F=4 versus PolderCast with F=2, high churn speed

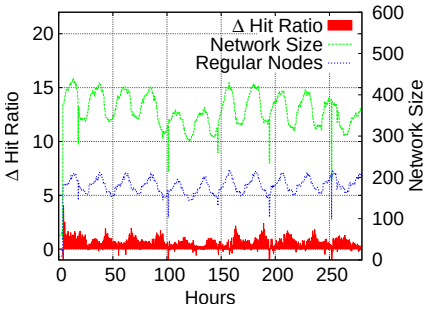
Figure 6.5: Hit-ratio improvement when using PolderCast with F=4 versus PolderCast with F=2

would be equivalent to what we are doing to increase the speed of playback of the churn trace in the simulations. This means that this new approach can be used to lower the overhead of the maintenance.

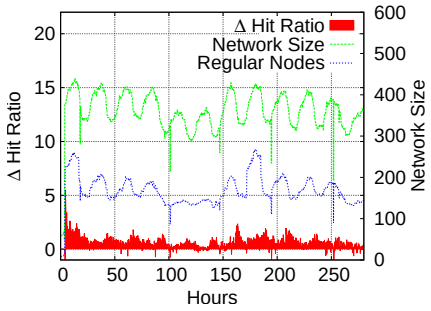
When comparing the daily CRB with the weekly one, we see that in the first week of the trace, both approaches look very similar in their improvements. However, after this, the data structures of the weekly regularity approach now contain correctly identified regular nodes and therefore we see the improvement for the second week of the trace to be higher than the first one. In general the weekly CRB seems to be more effective than the daily regularity one, especially for the second week.

The graphs give us just a general view of the improvements we can achieve by using the regularity approach. However we need to look into the figures behind the plots in greater detail. These are represented in Table 6.2, where the values are calculated for the entire length of the experiments, even though there is not enough space to put these into graphs in order to avoid confusion. Here we can also see the values that represent the experiments for 100 seconds as cycle period.

As we can see from this table, the more we increase the cycle period, the more we see the benefits of the regularity approach. PolderCast with $F = 2$ is outperformed by of 7-8% on average for the 1000 seconds cycle experiments. Maximum improvements are also interesting because as we can see from the graphs, it is especially in the periods when

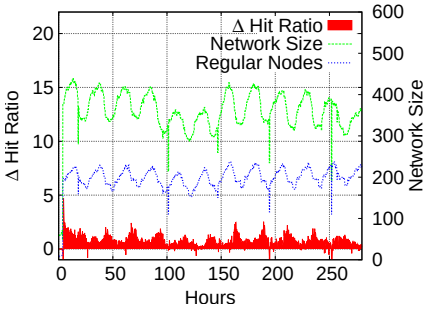


(a) Hit-ratio improvement when using regularity with γ=75% with daily period versus PolderCast with F=4, medium churn speed

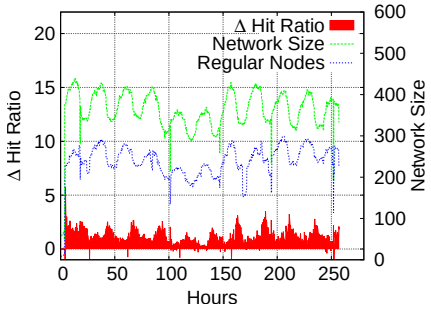


(b) Hit-ratio improvement when using regularity with γ=75% with weekly period versus PolderCast with F=4, medium churn speed

Figure 6.6: Hit-ratio improvement when using regularity with $\gamma = 75\%$, medium churn speed



(a) Hit-ratio improvement when using regularity with γ=70% with daily period versus PolderCast with F=4, medium churn speed

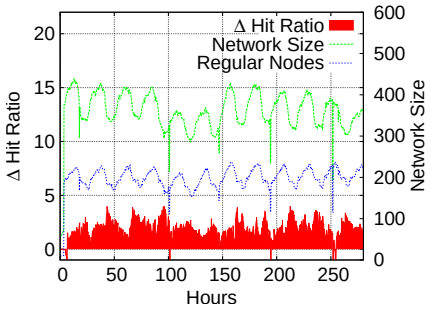


(b) Hit-ratio improvement when using regularity with γ=70% with weekly period versus PolderCast with F=4, medium churn speed

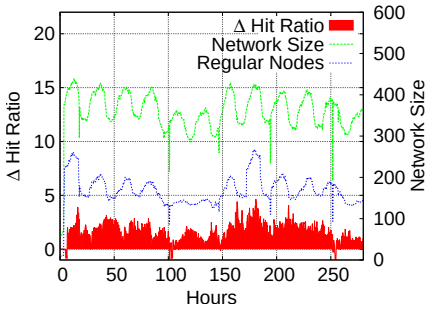
Figure 6.7: Hit-ratio improvement when using regularity with $\gamma = 70\%$, medium churn speed

a large number of nodes departs from the system, that this maximal improvement happens. These maximal improvements for the different regularity experiments are between 9-17% when compared to PolderCast with $F = 2$ or between 1-7% when compared to PolderCast with $F = 4$, and this improvement is quite significant. It should be noted that in the cases when we compare the 100 seconds cycle, or the 200 seconds one, by having a look also at Table 6.3, we see that PolderCast by itself does a very good job at achieving on average 97-99%. This means that there is not much room for improvement and any improvement over that is still considerable.

What is noticeable from Table 6.2 is that sometimes there is a deterioration in hit-ratio when using the regularity approach. This appears to happen even when comparing PolderCast $F = 4$ against the one with $F = 2$. The only logical explanation to this is that during the execution of the dissemination protocol, the order in which the different nodes receive and forward a message might result in a "chance" of leaving out some of the nodes from receiving the message. When comparing the regularity approach to $F = 4$ however, these negative results do not constitute the norm (as can be seen throughout the plots). Deteriorations in hit-ratio usually happen either in the very beginning of the experiments or during large node de-

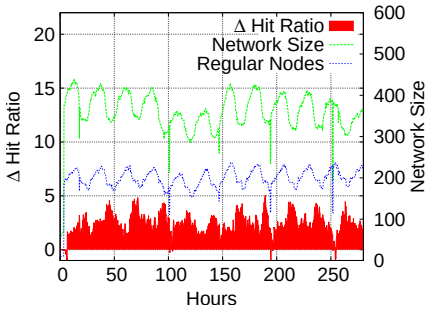


(a) Hit-ratio improvement when using regularity with $\gamma = 75\%$ with daily period versus PolderCast with F=4, high churn speed

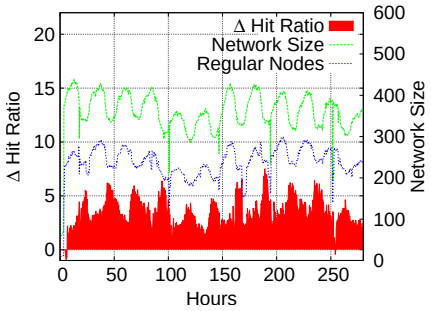


(b) Hit-ratio improvement when using regularity with $\gamma = 75\%$ with weekly period versus PolderCast with F=4, high churn speed

Figure 6.8: Hit-ratio improvement when using regularity with $\gamma = 75\%$, high churn speed



(a) Hit-ratio improvement when using regularity with $\gamma = 70\%$ with daily period versus PolderCast with F=4, high churn speed



(b) Hit-ratio improvement when using regularity with $\gamma = 70\%$ with weekly period versus PolderCast with F=4, high churn speed

Figure 6.9: Hit-ratio improvement when using regularity with $\gamma = 70\%$, high churn speed

parture events such as the ones around 100, 200 and 250 hours of trace. In these moments, we believe that PolderCast with $F = 4$ has better hit ratio due to the fact that it is using all of the nodes with double the amount of forwarding, while the regularity approach is using only the regular nodes to do this. However, the regular nodes population is also undergoing large departures in such events and this might have resulted in a lower hit ratio. In the beginning of the experiments the data structures of GRID and RingsRegular are not yet populated with regular nodes. Therefore it is natural that in this period PolderCast $F = 4$ has a better hit-ratio.

Another aspect to be discussed is the influence of the choice of the regularity threshold γ. We see from the Table 6.2, but also in Table 6.3, that the lower regularity threshold in these experiments has had higher success in improving the hit ratio, both for the daily and weekly periods of regularity. This could be attributed to the fact that by lowering γ, a higher number of nodes is perceived as regular and therefore a higher number of nodes aids in the dissemination. Nonetheless this must not be the case if we continue to lower γ as this will then lower the probability that the regular ring remains not broken during the entire length of the timeslot. However, this needs further investigation for each specific application that this approach might be used for. The main factor that would influence the

decision could be the frequency with which the messages are generated and the timeliness that the messages are required to be delivered.

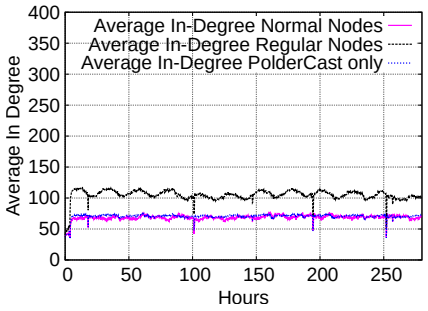
Table 6.2: Absolute hit ratio percentage difference between different protocols

	Improvement vs F=2			Improvements Vs. F=4		
Protocol	Avg	Min	Max	Avg	min	max
PolderCast 100 sec cycle F=4	0.4399	-0.1876	9.1213	n/a	n/a	n/a
PolderCast 200 sec cycle F=4	1.2000	-0.0909	9.8491	n/a	n/a	n/a
PolderCast 1000 sec cycle F=4	5.3335	0.0000	10.5730	n/a	n/a	n/a
Regularity Daily 70% 100 sec cycle	0.6773	-1.7168	11.3453	0.2374	-3.9091	4.9933
Regularity Daily 75% 100 sec cycle	0.6417	-2.6547	10.4316	0.2017	-5.3859	4.0796
Regularity Daily 70% 200 sec cycle	1.8502	-2.3405	11.6303	0.6502	-3.5573	4.7020
Regularity Daily 75% 200 sec cycle	1.7363	-2.3926	11.3085	0.5363	-4.7036	4.0686
Regularity Daily 70% 1000 sec cycle	7.6623	-0.4318	15.1236	2.3288	-2.1452	5.5954
Regularity Daily 75% 1000 sec cycle	7.1811	-0.5941	14.3163	1.8475	-3.6389	4.5306
Regularity Weekly 70% 100 sec cycle	0.8516	-0.9211	12.3460	0.3776	-1.8421	5.9940
Regularity Weekly 75% 100 sec cycle	0.6365	-1.0183	6.9949	0.2052	-1.5967	1.9629
Regularity Weekly 70% 200 sec cycle	2.4009	-1.9284	12.8393	1.0449	-2.3432	5.7512
Regularity Weekly 75% 200 sec cycle	1.4516	-1.5102	10.3734	0.4221	-3.3173	2.5652
Regularity Weekly 70% 1000 sec cycle	8.6488	-0.8235	17.7482	3.3153	-0.9297	7.4826
Regularity Weekly 75% 1000 sec cycle	6.9634	-0.6452	14.5112	1.6299	-1.9251	4.6392

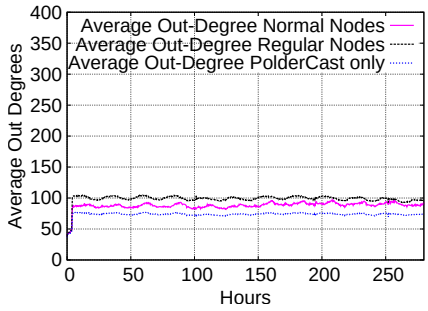
6.7.2 Node Degree

Our approach is adding additional protocols on top of the ones that are part of PolderCast. This implies that we expect the degrees of the nodes to be higher compared to PolderCast alone since the nodes are creating more connections. The values for nodes' in, out and total degree are presented in Table 6.3. We have also shown the time-series of network average in-degree and out-degree for a run of the regularity approach in Figure 6.10.

In Figure 6.10 we see that the average of the in-degrees of all normal nodes is similar to the in-degree nodes have in PolderCast. This is to be expected since these nodes are not chosen by other nodes as neighbors since they are not regular. Regular nodes on the other hand have almost double the value of in-degree when compared to normal nodes or PolderCast. This is because other regular nodes prefer these nodes as neighbors for their gossiping protocols. The fact that the in-degree of regular nodes fluctuates in sync with the population of regular nodes is also because it is regular nodes that choose other regular nodes as gossiping partners. The value is almost doubled because of the fact that the RingsRegular layer degree is proportional to the number of topics a node is subscribed since there is



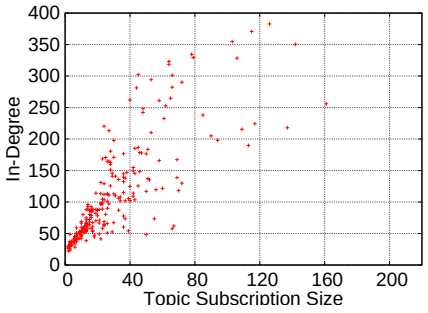
(a) In-Degree network average time series for daily regularity $\gamma = 70\%$, compared with PolderCast. Slow churn speed.



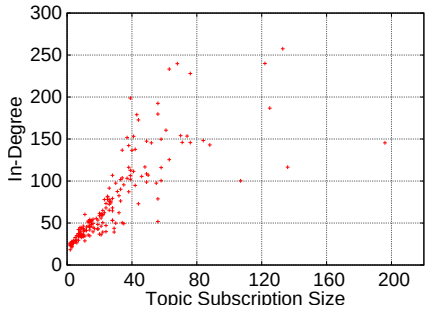
(b) Out-Degree network average time series for daily regularity $\gamma = 70\%$, compared with PolderCast. Slow churn speed.

Figure 6.10: Node in and out degrees averages time series, compared with PolderCast

one ring per topic. Therefore adding another layer of rings for the regular nodes, the in and out degrees of these nodes are almost doubled when compared to the in and out degrees of nodes in PolderCast.



(a) In-Degree distribution based on topic subscriptions of regular nodes only.

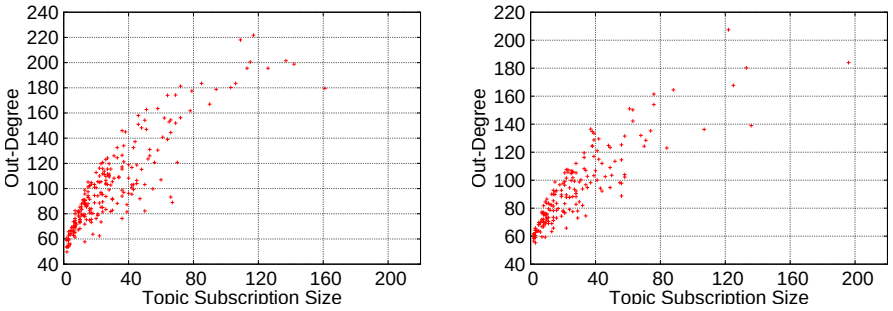


(b) In-Degree distribution based on topic subscriptions of normal nodes only

Figure 6.11: Plot showing relationship between topic subscription size and in-degree of regular and normal nodes. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed.

For the out degrees in Figure 6.10b, the same trend as for the in-degrees is seen for the regular nodes. The normal nodes on the other hand have a higher out-degree than the nodes in PolderCast. This is explained by the fact that Algorithm 8 does not distinguish between the cases when the node running it is regular or not. This means that even normal nodes try to maintain $V_{\text{RingsRegular}}[\text{currentTimeslot}]$ even though they are not regular in the current timeslot. This is done so that in case these nodes become regular in any timeslot where they previously were not, they already know some potential regular neighbors in the regular ring. As a downside, this increases the out-degree of the non-regular nodes as they have to maintain more connections. The in-degree of these nodes is not affected since regular nodes do not connect to them because they are not part of the regular ring.

The in and out degrees of nodes are affected by the size of the topic subscriptions that the nodes have. For this reason we plotted the relationship between degrees and topic subscriptions of the regular nodes and normal ones in Figure 6.11 and 6.12. In these figures we are showing

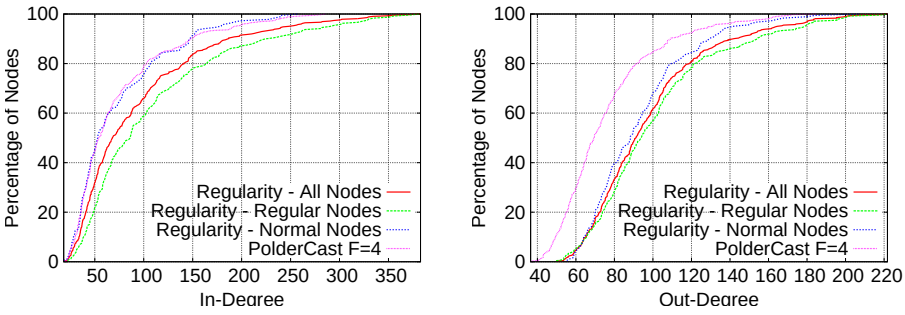


(a) Out-Degree distribution based on topic subscriptions of regular nodes only

(b) Out-Degree distribution based on topic subscriptions of normal nodes only

Figure 6.12: Plot showing relationship between topic subscription size and out-degree of regular and normal nodes. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed.

only results for daily r-egularity with $\gamma = 70\%$, stopped around hour 158 where the number of nodes in the system is at its peak comparing to the whole simulation. The relationship between number of topics the node is subscribed to and its degree is not very clear since the presence of churn does not permit for the overlay to have converged to stable values. However, the general trend shown also from Figure 6.10 is also visible here. The regular nodes have an in-degree which is generally 2-4 times the size of topic subscriptions, while the normal nodes usually stay in the vicinity of values that are 2 times the size of topic subscriptions. For out-degrees we see very similar values between normal nodes and regular nodes (Figure 6.12), both in the vicinity of 4 times the size of topic subscriptions. Nonetheless, the normal nodes have lower values.



(a) In-Degree CDF

(b) Out-Degree CDF

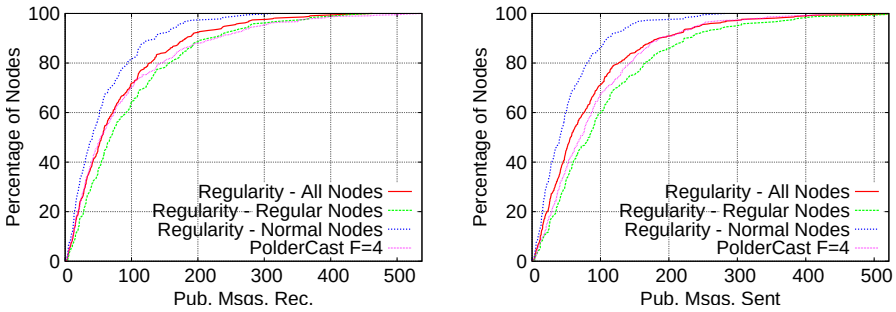
Figure 6.13: CDFs of in and out degrees in regularity runs, compared also with PolderCast. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed.

In Figure 6.13 we see the cumulative distribution function (CDF) plots of the in-degree and out-degree for normal and regular nodes, compared also with PolderCast. These plots serve to prove the points mentioned above about the in and out degrees of the nodes, however it does not introduce any new findings.

6.7.3 Number of messages sent and received

Publication messages

The overall results with regards to the number of messages sent (tx) and received (rx) is shown in Table 6.3 in columns 2-3 for the overall average and the columns 9-12 for the normal nodes and regular nodes specifically. As could very well be predicted from how we described our approach would work, the number of messages sent from regular nodes is roughly double the amount of messages sent and received by the normal nodes. When taking an overall average, we see that the average number of messages sent and received is not increasing by much, being still less than the number of messages the nodes send in the cases of PolderCast with $F = 4$.



(a) Publication messages received CDF

(b) Publication messages sent CDF

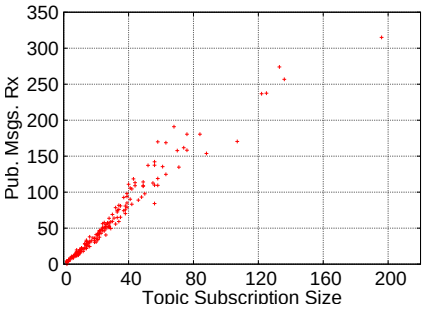
Figure 6.14: CDFs of received and sent pub. msgs. in regularity runs, compared also with PolderCast. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed.

In order to see in more detail how the distribution of the number of publication messages is across the nodes in the system, we have plotted the CDF incoming and outgoing messages in Figure 6.14. For comparison we have also included the distribution in the case of PolderCast with F=4.

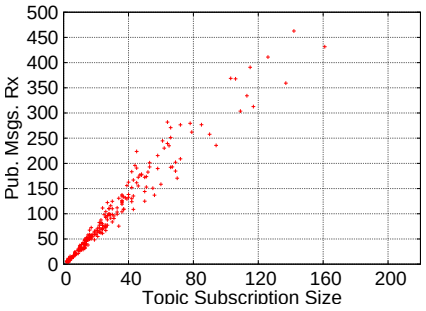
For incoming messages in Figure 6.14a we see that overall normal nodes (blue line) receive half of the messages compared to regular nodes (green line). On the overall view, 75% of all nodes in the regularity run (red line) receive almost the same amount of messages as PolderCast nodes (purple line), while the rest deviate, receiving less messages.

For the outgoing messages in Figure 6.14b, we see again the same trend where normal nodes send about half the number of the messages that the regular nodes send. On the overall view, the number of messages sent follows almost the same distribution as PolderCast with F=4, with the difference that around 90% of the nodes send less messages.

The number of publication messages sent and received is dependent on the number of topics the nodes are subscribed to. For this reason we have plotted the dependency between topic subscription size and number of messages received in Figure 6.15 and sent in Figure 6.16. We have separately plotted the relationship for regular nodes and for normal ones. For both the sent and the received number of messages, the relationship is clearly linear.



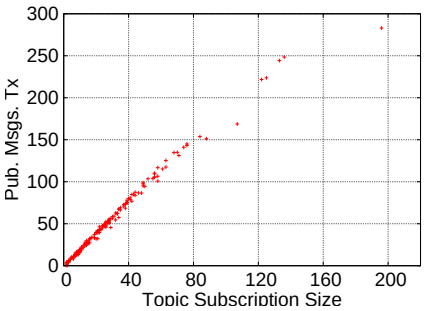
(a) Topic subscription size and publishing messages received relationship for normal nodes



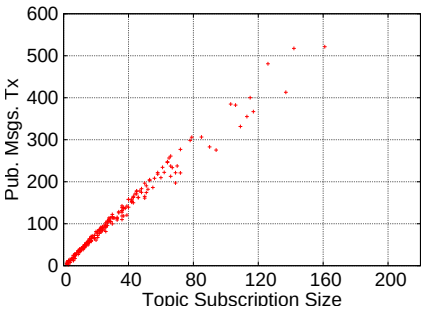
(b) Topic subscription size and publishing messages received relationship for regular nodes

Figure 6.15: Topic subscription size and publishing messages received relationship. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed.

For the number of messages received, the normal nodes seem to receive roughly 2 messages per topic subscribed to. Regular nodes seem to have a coefficient between 3 and 4. This can be verified from sub-figures in Figure 6.15. While, for the number of messages sent, it is as expected, with normal nodes sending around 2 messages for each topic they are subscribed to, while the regular ones 4.



(a) Topic subscription size and publishing messages sent relationship for normal nodes



(b) Topic subscription size and publishing messages sent relationship for regular nodes

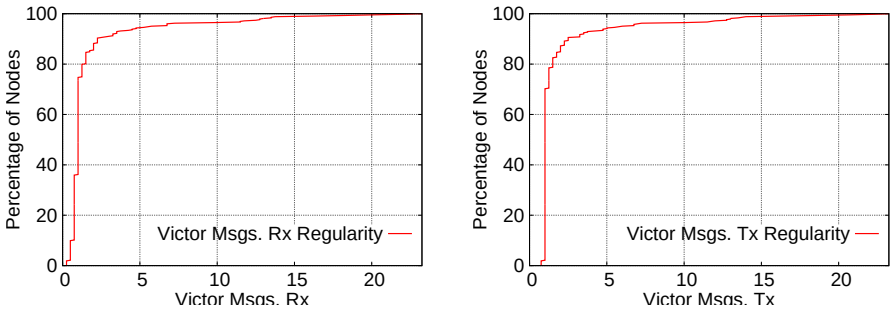
Figure 6.16: Topic subscription size and publishing messages sent relationship. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed.

The unbalance between the number of messages the regular nodes send and receive and the normal nodes do is due to the fact that the regular nodes disseminate in both the regular rings and the normal ones. Therefore they also receive messages in both rings. This could be argued as unfair towards the nodes which exhibit regularity, however we explained earlier that this was to be expected when trying to take advantage of a property of a subset of the nodes.

Overhead messages: Victor & RingsRegular

In this section we will take a look at the overhead messages resulting from gossiping at the Victor and RingsRegular protocols, which are the protocols that we have additionally when compared with PolderCast. For

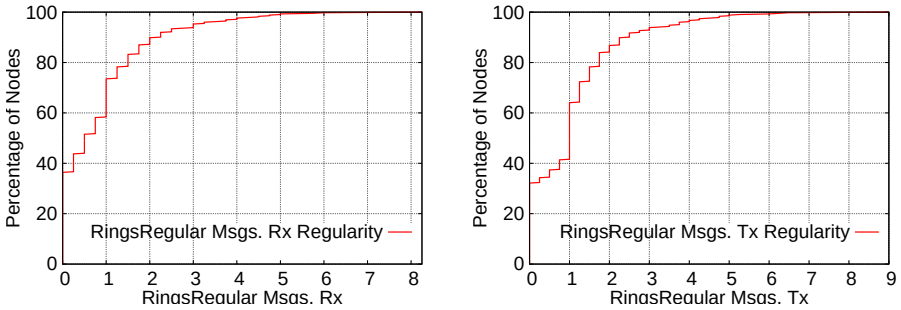
this purpose we have plotted a CDF of the messages received and sent in Victor layer in Figure 6.17 and in Figure 6.18 for RingsRegular. Also we have put overall average values during the whole execution in Table 6.3 in columns 4-5.



(a) Victor messages received CDF (b) Victor messages sent CDF

Figure 6.17: CDFs of received and sent gossiping messages in Victor. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed.

For Victor we can see that 90% of the nodes receive less than 3 messages, while the other 10% receives between 3 and 25 messages. This could be for the reason that not all nodes are regular and the normal nodes are not preferred as gossiping partners in Victor, unless it is a response to a gossip request. The long tail distribution shows that a low number of nodes face a higher load in gossiping. The same results are also visible for the messages sent in Victor, with the exception that most of the nodes in this case send 1 message since they all have to send one gossip request per cycle, while few nodes send more messages as they have to reply to requests from other nodes.



(a) RingsRegular messages received CDF (b) RingsRegular messages sent CDF

Figure 6.18: CDFs of received and sent gossiping messages in RingsRegular. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed.

For the RingsRegular layer, we see that nodes receive and send between 0 and 7 messages. The number of messages is few compared to Victor because of the ring structure in which nodes gossip with their closest neighbors in the ring, which are the nodes that have their IDs closest to their own. The fact that normal nodes are not part of the regular rings is also shown in the large number of nodes receiving and sending 1 or less messages. The RingsRegular layer seems fairly balanced. Besides that,

there are no other interesting observations to be made for this layer.

6.7.4 Path length

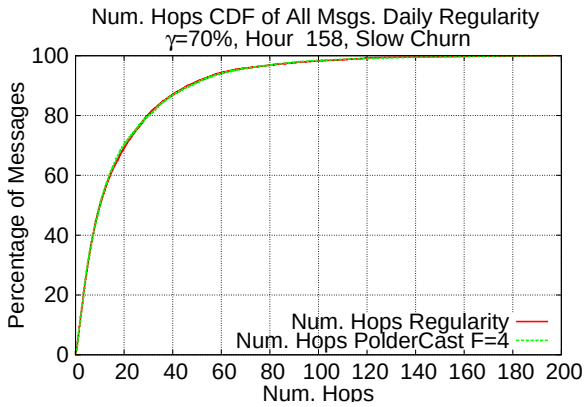


Figure 6.19: Number of Hops CDF of all messages sent. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed.

The path length is represented by the average number of hops a message goes to before being delivered for the first time to a node in Table 6.3 on the first column. The overall trend that can be seen from the numbers in this column is the fact that by using the regularity approach, any of the experiments run with it, the path lengths are almost the same as PolderCast with F=4.

In order to look more into this metric we also plotted the CDF of number of hops a message goes through before being delivered for the first time to a node interested in it. This plot is shown in Figure 6.19. From all of these plots we see that the behavior of the regularity run is almost identical to PolderCast with F=4.

6.7.5 Message Redundancy

As we explained above, this parameter represents overhead as it has no benefit to the dissemination. It is presented in the 11-th column of Table 6.3. We see clearly from the numbers in this column that the overall message duplication for the various experiments with the regularity approach is higher than the values of this parameter for the PolderCast with $F = 2$ but smaller than those for PolderCast with $F = 4$. This is to be expected as not all the nodes perform double dissemination in the case of the regularity approach. However, it is still promising considered that the overall performance in the aspect of dissemination outperforms PolderCast with $F = 4$.

In order to look more into this metric we also plotted a CDF of the redundancy of all messages in Figure 6.20. From this figure we see that the regularity approach creates less duplications than PolderCast with F=4 for about 50% of the messages.

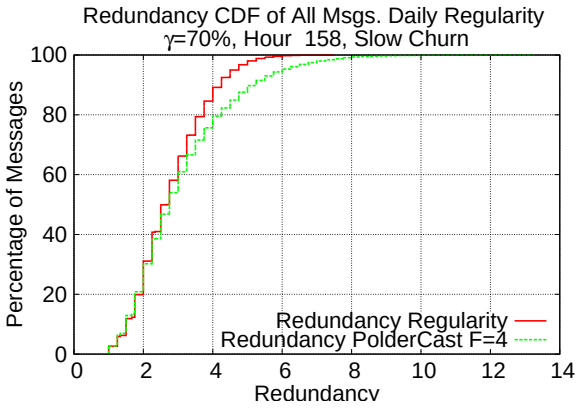


Figure 6.20: Redundancy CDF - All Messages. Results for daily regularity $\gamma = 70\%$, hour 158, slow churn speed.

6.7.6 Possible improvements to evaluation in the future

In the evaluation of RingsRegular we showed that there is improvement in hit-ratio when compared with PolderCast. The hit-ratio we measured is the average hit-ratio for all of the topics in the system. In the future it might be interesting to look at the relationship between topic popularity and other metrics, such as hit-ratio, path length and message redundancy. Such an evaluation can show how fair both PolderCast alone and PolderCast with RingsRegular are with regards to various topics with different popularities.

Table 6.3: Table of all other parameters used to benchmark the new protocol

		Avg. Num. Msgs. Publication		Avg. Num. Msgs. Victor & RingsRegular		Avg. Degree All Nodes			Avg. Num. Msg.Publication Normal Only		Avg. Num. Msg.Publication Regular Only		Avg. Degree NormalOnly			Avg. Degree RegularOnly				
Protocol	**Avg. Num. Hops**	**Rx**	**Tx**	**Rx**	**Tx**	**In**	**Out**	**Total**	**Rx**	**Tx**	**Rx**	**Tx**	**In**	**Out**	**Total**	**In**	**Out**	**Total**	**Dupl. Fact.**	**Hit Ratio**
PolderCast 100 sec cycle F=2	8.1908	51.7034	52.5811	n/a	n/a	70.4137	73.6287	108.9901	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	1.7910	98.9314
PolderCast 100 sec cycle F=4	8.2995	85.3698	87.2687	n/a	n/a	70.4063	73.6280	108.9906	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	2.7906	99.3713
PolderCast 200 sec cycle F=2	7.9027	47.6420	51.0145	n/a	n/a	68.2759	73.9444	108.8819	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	1.7004	97.2123
PolderCast 200 sec cycle F=4	8.1954	81.8190	87.2322	n/a	n/a	68.2701	73.9463	108.8830	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	2.6879	98.4123
PolderCast 1000 sec cycle F=2	6.1644	32.8638	40.6193	n/a	n/a	59.6908	73.2294	106.1103	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	1.4749	88.9933
PolderCast 1000 sec cycle F=4	7.6585	67.7088	83.5979	n/a	n/a	59.6985	73.2122	106.0892	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	2.3395	94.3268
Regularity Daily 70% 100 Sec cycle	8.3163	78.1216	82.4410	2.6277	3.0333	88.2576	93.7334	134.0237	56.5008	52.0656	93.9370	104.5482	68.4429	88.8779	103.3584	102.9030	97.4487	120.7800	2.6539	99.6087
Regularity Daily 75% 100 Sec cycle	8.3104	75.8201	79.3396	2.6693	3.0129	88.9740	94.0406	134.7420	56.0668	52.5392	93.8375	103.7091	69.4244	89.3788	105.9944	107.0033	98.4252	119.5742	2.5871	99.5730
Regularity Daily 70% 200 Sec cycle	8.2348	75.0437	82.2194	2.6823	3.0651	86.1230	94.2072	134.3235	52.7868	51.3221	91.3090	104.7089	65.7228	89.2825	103.5401	101.2277	97.9641	121.2864	2.5638	99.0625
Regularity Daily 75% 200 Sec cycle	8.2192	72.6447	79.0296	2.7298	3.0443	86.9251	94.6123	135.1855	52.3874	51.7934	91.1124	103.7968	66.7476	89.9989	106.3570	105.5172	98.9668	120.1346	2.4959	98.9485
Regularity Daily 70% 1000 Sec cycle	7.7858	64.6219	79.4245	2.9755	3.2451	78.8962	94.8172	134.0364	40.8326	46.2759	82.0266	103.5780	54.6399	89.2858	102.3259	96.7981	99.0347	122.2235	2.3029	96.6556
Regularity Daily 75% 1000 Sec cycle	7.7091	62.0715	75.8657	3.0004	3.2108	79.4798	94.9714	134.6367	40.5149	46.5937	81.7475	102.5052	55.8455	89.6738	104.6517	101.1849	99.9591	121.1016	2.2385	96.1743
Regularity Weekly 70% 100 sec cycle	8.6129	80.8637	84.6745	3.2301	3.4371	86.6775	91.3580	131.2258	56.6251	51.0710	94.2366	103.2997	67.4809	83.8262	96.1052	97.6501	95.5476	122.0451	2.7448	99.6625
Regularity Weekly 75% 100 sec cycle	8.7254	74.8122	76.6173	3.2212	3.2716	88.1853	91.8434	133.0362	54.2685	51.4158	98.0349	104.9937	68.6701	85.2666	103.8484	110.6659	99.3278	119.6698	2.5454	99.5285
Regularity Weekly 70% 200 sec cycle	8.6813	79.0139	86.1289	3.2731	3.4797	85.4155	92.9252	132.9675	54.2370	51.6085	91.9081	104.0535	65.6159	86.4857	98.2121	96.0275	96.4123	123.5958	2.6811	99.1883
Regularity Weekly 75% 200 sec cycle	8.2557	72.4229	76.8740	3.3303	3.3694	86.4528	92.4450	132.6931	51.0258	50.7339	95.7977	105.2937	65.9995	86.3068	104.1032	109.1416	99.2474	119.2381	2.4828	98.9357
Regularity Weekly 70% 1000 sec cycle	7.9449	70.3816	85.7434	3.4605	3.6162	78.5662	93.8758	132.0954	43.2273	47.7015	82.7179	102.9191	54.5500	87.6813	97.4860	89.6724	96.8765	123.9383	2.4661	97.6420
Regularity Weekly 75% 1000 sec cycle	7.7003	62.1687	74.1438	3.4015	3.4251	79.5716	93.4258	132.8100	39.8191	45.8975	85.7889	103.8673	55.8962	86.7106	102.7812	104.8720	100.7084	120.9740	2.2376	95.9567

6.8 Conclusions

In conclusion:

> **Hit-ratio Improvement**
>
> We improve the hit ratio in PolderCast by using regularity and we demonstrated it. The regularity approach outperforms PolderCast with $F = 4$ with regards to hit ratio, while having a slightly lower message duplication and similar path lengths. The price is some higher overhead in terms of overlay maintenance messages, higher node degree and slightly higher bandwidth consumption, however we show that this is not very significant.

The improvements to hit ratio seem to be higher with an increase in the period between gossip cycles, so applications that are especially sensitive to bandwidth and which do not request very frequent communication between nodes could take advantage of this approach. The improvements do not seem to be dependent much on the fact that the CRB is weekly or daily, however the weekly one has slightly better results, especially after the first occurrence of the CRB. However, it will have higher memory and CPU consumption on each of the nodes for its data-structures. Also, the regularity threshold γ needs to be carefully configured according to the application and observed patterns in the system. A too high γ yields a low number of regular nodes which might not be enough to properly aid into the dissemination, while a too low γ (always bigger than 50%) might give nodes that are not regular enough to provide a reliable dissemination in the regular ring. There is no best γ that will work for all the various applications. For our application, from the experiments we conducted, $\gamma = 70\%$ was the most appropriate.

We showed that the duplication of messages is lower in the regularity approaches, compared to PolderCast with $F = 4$. At the same time we showed that the path length of messages from source to destination is no different from the one of PolderCast with $F = 4$. These statements are true regardless of the settings for regularity (i.e. γ, timeslot duration, CRB duration). So there is no deterioration resulting from the regularity approach with regards to these metrics.

On the downside, this particular new approach increases the overall nodes' degree, almost doubling it at times. This could impact the performance of the applications built around such a protocol as they would have to maintain a higher number of live connections. Another negative aspect of the regularity approach is that there is a higher load on the regular nodes for the benefits of the whole system. The regular nodes send and receive double the number of dissemination messages that the other nodes do, therefore putting higher processing and bandwidth requirements on them. In a real world system this might discourage users who exhibit regular behavior from using the system. However, there is no way of exploiting a property of a subset of the nodes without putting some more load on them.

The solution presented here as a proof of concept for using GRID and the regularity information of nodes in order to improve dissemination in a topic-based publish-subscribe system has shown that some improvements are achievable, while some trade-offs have to be made. This solution is designed specifically for the way PolderCast works and therefore it does not apply to other P2P topic-based pub-sub systems with different types of overlays.

This proof-of-concept implementation was not only useful towards demonstrating that the regularity concept can be used to improve the hit ratio of PolderCast, but also that GRID is configurable and generic enough to be utilized for this application. It is our belief that it is generic enough to be used by other applications that want to take advantage of the regularity property of nodes.

Chapter 7

Summary and Conclusions

Churn characteristics at the individual node level in P2P systems and how to possibly extract this information and how to take advantage of it in existing P2P topic-based publish-subscribe systems, has been the focus of this thesis. In recent years, P2P systems have gained in popularity and many different systems are being built with this approach. Publish-subscribe model is one of the most popular paradigms for implementing many of the services that are now coming to define the web. Therefore the combination of these two topics together is gaining a lot of attention from researchers.

The first topic we touched in this thesis is the identification of a property of churn that allows systems to somehow predict the behavior of nodes in the near and distant future. By using statistical information about the nodes up and down times for two fairly popular systems such as Skype and KAD, we tried to identify and show the parameters that affect the number of nodes that are perceived as having a regular behavior. This type of behavior implies that the nodes on-line and off-line periods follow an individual pattern. Therefore there is reason to believe, as shown by other research, that this behavior will continue in the future, allowing us to roughly predict when a specific node is going to be on-line next, with a certain probability. We showed how to gather such regularity information and which parameters influence the number of nodes perceived as regular. We showed that this is dependent on the regularity threshold, while it is not dependent on the timeslot duration. Later in the thesis we show that the CRB duration is also important. Our experiments indicate that weekly CRB offers more accuracy in predicting individual node behavior.

Secondly we designed GRID, a generic service for identifying the node's own regularity and discovering other regular nodes based on similar regularity, combined with interests or another distance metric provided by the application. This service is designed to be lightweight and robust, following a modular approach with two layers, a random overlay protocol such as Cyclon and a clustering layer that is a modification of Vicinity for regularity called Victor. As part of this solution we also explained how regularity information can be exchanged between nodes in a bandwidth efficient manner of one bit for timeslot and similar to the way

nodes exchange topic subscriptions in topic-based pub-sub systems.

The third topic we researched in this thesis is how the regularity information about the nodes can be used in order to improve the dissemination of P2P topic-based publish-subscribe systems. For this reason we designed and implemented an extension on an already existing system called PolderCast. In this part we used GRID as a building block of our solution, to prove that it can be configured to serve different applications. We implemented the extensions in the PeerSim simulator and then performed tests with various regularity related parameters and different churn rates. We showed from the results of the experiments that using the regularity approach improves the overall dissemination efficiency considerably, even compared to PolderCast with a double fanout factor. We showed that there were increases in hit-ratio, lower message duplication and similar path lengths. On the downside, there were also negative effects such as increased unfairness, meaning more messages were sent and received by the regular nodes. At the same time the degree of the nodes was increased, especially that of regular nodes. We explore the different parameters related to regularity. We conclude that for our implementation, the weekly regularity version has slightly higher improvement in hit-ratio compared to the daily one. The best improvements in hit-ratio are noticed when the churn is at high rates.

These results are promising in terms of showing that the regularity information of the nodes could be used to improve existing overlays in terms of dissemination in case they are going to be used for applications which need short delays between message creation and delivery. Also since we have shown that we could achieve the same results as PolderCast with 10 times less frequent overhead overlay maintenance messages, this solution could also be used in applications that request infrequent communication. This situation is the same as a system where churn is very high, so the regularity approach could also be a viable alternative in such systems.

Chapter 8

Future Work

We introduced GRID, the generic service that we created for identifying the node's own regularity pattern, exchanging it with other nodes and discovering other regular nodes close to the service consumer needs. We provided one proof-of-concept application called RingsRegular that makes use of GRID and in the future we would like to explore other applications that make use of it.

While the results regarding the usage of regularity information to improve message dissemination in P2P topic-based publish-subscribe systems presented in this thesis might be promising, the application RingsRegular we presented is PolderCast specific. In the future it would be interesting to try using node regularity with GRID as a building block in other systems like Scribe or even constructing a new type of overlay centered around the regularity concept.

In addition, we would like to test GRID separately with various implementations of the callback interface. This will allow us to see how generic the service is and its limitations for certain applications.

Internet usage is changing everyday as the Internet is becoming more and more accessible all over the world and time spent on-line for the younger generations of people is ever increasing. Therefore arises the need to study contemporary systems. In this paper we were limited in the amount of traces already available from other works. However, longer and more recent traces could reveal new patterns.

For the experiments presented in Section 6.7 we used the Skype trace. It would be interesting to run experiments with the KAD trace or other traces that might be available in the future. This would help us understand how the approach we presented in Chapter 6 would help to improve dissemination in systems with different ratio of regular nodes to total on-line nodes population than Skype.

Bibliography

[1] Sebastien Baehni, Patrick Th Eugster and Rachid Guerraoui. 'Data-aware multicast'. In: *Dependable Systems and Networks, 2004 International Conference on*. IEEE. 2004, pp. 233–242.

[2] Roberto Baldoni, Roberto Beraldi, Vivien Quema, Leonardo Querzoni and Sara Tucci-Piergiovanni. 'TERA: topic-based event routing for peer-to-peer architectures'. In: *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*. ACM. 2007, pp. 2–13.

[3] R. Bhagwan, S. Savage and G. Voelker. 'Understanding Availability'. In: *Proceedings of IPTPS'03*. 2003.

[4] Ranjita Bhagwan, Stefan Savage and Geoffrey M. Voelker. *Understanding Availability*. 2003.

[5] W. Bolosky, J. Douceur, D. Ely and M. Theimer. 'Feasibility of a Serverless Distributed file System Deployed on an Existing Set of Desktop PCs'. In: *Proceedings of SIGMETRICS*. 2000.

[6] Antonio Carzaniga, David S Rosenblum and Alexander L Wolf. 'Achieving scalability and expressiveness in an internet-scale event notification service'. In: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. ACM. 2000, pp. 219–227.

[7] Miguel Castro, Peter Druschel, A-M Kermarrec and Antony IT Rowstron. 'SCRIBE: A large-scale and decentralized application-level multicast infrastructure'. In: *Selected Areas in Communications, IEEE Journal on* 20.8 (2002), pp. 1489–1499.

[8] Gregory Chockler, Roie Melamed, Yoav Tock and Roman Vitenberg. 'SpiderCast: a scalable interest-aware overlay for topic-based pub/sub communication'. In: *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*. DEBS '07. New York, NY, USA: ACM, 2007, pp. 14–25. ISBN: 978-1-59593-665-3. DOI: 10.1145/1266894.1266899. URL: http://doi.acm.org/10.1145/1266894.1266899.

[9] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco and Gianpaolo Cugola. 'Introducing reliability in content-based publish-subscribe through epidemic algorithms'. In: *Proceedings of the 2nd international workshop on Distributed event-based systems*. ACM. 2003, pp. 1–8.

[10] Paolo Costa and Gian Pietro Picco. 'Semi-probabilistic content-based publish-subscribe'. In: *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*. IEEE. 2005, pp. 575–585.

[11] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. ISBN: 0132143011, 9780132143011.

[12] Matteo Dell'Amico, Pietro Michiardi and Yves Roudier. 'Back To The Future: On Predicting User Uptime'. In: *CoRR* (2010), pages.

[13] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui and Anne-Marie Kermarrec. 'The many faces of publish/subscribe'. In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: http://doi.acm.org/10.1145/857076.857078.

[14] Jim Farley. *Java distributed computing*. O'reilly, 1998.

[15] Ayalvadi J Ganesh, Anne-Marie Kermarrec and Laurent Massoulié. 'SCAMP: Peer-to-peer lightweight membership service for large-scale group communication'. In: *Networked Group Communication*. Springer, 2001, pp. 44–55.

[16] Simson L Garfinkel. 'VoIP and Skype security'. In: *Tactical Technology Collective* 12 (2005).

[17] Sarunas Girdzijauskas, Gregory Chockler, Ymir Vigfusson, Yoav Tock and Roie Melamed. 'Magnet: practical subscription clustering for Internet-scale publish/subscribe'. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. DEBS '10. Cambridge, United Kingdom: ACM, 2010, pp. 172–183. ISBN: 978-1-60558-927-5. DOI: 10.1145/1827418.1827456. URL: http://doi.acm.org/10.1145/1827418.1827456.

[18] Saikat Guha and Neil Daswani. *An experimental study of the skype peer-to-peer voip system*. Tech. rep. Cornell University, 2005.

[19] Krishna P Gummadi, Stefan Saroiu and Steven D Gribble. 'King: Estimating latency between arbitrary internet end hosts'. In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM. 2002, pp. 5–18.

[20] Márk Jelasity and Ozalp Babaoglu. 'T-Man: Gossip-based overlay topology management'. In: *In 3rd Int. Workshop on Engineering Self-Organising Applications (ESOA'05*. Springer-Verlag, 2005, pp. 1–15.

[21] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec and Maarten van Steen. 'Gossip-based peer sampling'. In: *ACM Trans. Comput. Syst.* 25.3 (2007), p. 8. ISSN: 0734-2071.

[22] Anne-Marie Kermarrec, Erwan Le Merrer, Gilles Straub and Alexandre Van Kempen. 'Availability-based methods for distributed storage systems'. Anglais. In: *SRDS 2012, 31st International Symposium on Reliable Distributed Systems*. Irvine, California., États-Unis, Oct. 2012. URL: http://hal.archives-ouvertes.fr/hal-00521034.

[23] Derrick Kondo, Bahman Javadi, Alexandru Iosup and Dick Epema. 'The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems'. In: *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE. 2010, pp. 398–407.

[24] Haewoon Kwak, Changhyun Lee, Hosung Park and Sue Moon. 'What is Twitter, a social network or a news media?' In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 591–600.

[25] Stevens Le Blond, Fabrice Le Fessant and Erwan Le Merrer. *Finding Good Partners in Availability-aware P2P Networks*. Anglais. Tech. rep. RR-6795. INRIA, 2009, p. 17. URL: http://hal.inria.fr/inria-00352529.

[26] Miguel Matos, Ana Nunes, Rui Oliveira and José Pereira. 'StAN: exploiting shared interests without disclosing them in gossip-based publish/subscribe'. In: *Proceedings of the 9th international conference on Peer-to-peer systems*. IPTPS'10. San Jose, CA: USENIX Association, 2010, pp. 9–9. URL: http://dl.acm.org/citation.cfm?id=1863145.1863154.

[27] Petar Maymounkov and David Mazieres. 'Kademlia: A peer-to-peer information system based on the xor metric'. In: *Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[28] James W. Mickens and Brian D. Noble. 'Exploiting availability prediction in distributed systems'. In: *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*. NSDI'06. San Jose, CA: USENIX Association, 2006, pp. 6–6. URL: http://dl.acm.org/citation.cfm?id=1267680.1267686.

[29] James W Mickens and Brian D Noble. 'Exploiting availability prediction in distributed systems'. In: *Ann Arbor* 1001 (2006), p. 48103.

[30] Alberto Montresor and Márk Jelasity. 'PeerSim: A Scalable P2P Simulator'. In: *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*. Seattle, WA, Sept. 2009, pp. 99–100.

[31] G. Muhl, L. Fiege and P. R. Pietzuch. *Distributed Event-based Systems*. Heidelberg: Springer-Verlag, 2006. ISBN: 978-3-540-32653-3.

[32] Alessio Pace, Vivien Quema and Valerio Schiavoni. 'Exploiting Node Connection Regularity for DHT Replication'. In: *Reliable Distributed Systems, IEEE Symposium on* (2011), pp. 111–120. ISSN: 1060-9857. DOI: http://doi.ieeecomputersociety.org/10.1109/SRDS.2011.22.

[33] Jay A Patel, Étienne Rivière, Indranil Gupta and Anne-Marie Kermarrec. 'Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems'. In: *Computer Networks* 53.13 (2009), pp. 2304–2320.

[34] Fatemeh Rahimian, Sarunas Girdzijauskas, Amir H. Payberah and Seif Haridi. 'Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish/Subscribe Enabling Rendezvous Routing in Unstructured Overlay Networks'. In: *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 746–757. ISBN: 978-0-7695-4385-7. DOI: 10.1109/IPDPS.2011.75. URL: http://dx.doi.org/10.1109/IPDPS.2011.75.

[35] Stefan Saroiu, P Krishna Gummadi and Steven D Gribble. 'Measurement study of peer-to-peer file sharing systems'. In: *Electronic Imaging 2002*. International Society for Optics and Photonics. 2001, pp. 156–170.

[36] Vinay Setty, Maarten van Steen, Roman Vitenberg and Spyros Voulgaris. 'PolderCast: fast, robust, and scalable architecture for P2P topic-based pub/sub'. In: *Proceedings of the 13th International Middleware Conference*. Middleware '12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 271–291. ISBN: 978-3-642-35169-3. URL: http://dl.acm.org/citation.cfm?id=2442626.2442644.

[37] Gyuwon Song, Suhyun Kim, Sunghwan Jang and Daeil Seo. 'Understanding Individual Nodes in Peer-to-Peer Systems'. In: *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*. Aug. 2010, pp. 1–6. DOI: 10.1109/ICCCN.2010.5560079.

[38] Gyuwon Song, Suhyun Kim, Sunghwan Jang and Daeil Seo. 'Understanding Individual Nodes in Peer-to-peer Systems'. In: *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*. IEEE. 2010, pp. 1–6.

[39] Gyuwon Song, Suhyun Kim and Daeil Seo. 'Replica Placement Algorithm for Highly Available Peer-to-Peer Storage Systems'. In: *Proceedings of the 2009 First International Conference on Advances in P2P Systems*. AP2PS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 160–167. ISBN: 978-0-7695-3831-0. DOI: 10.1109/AP2PS.2009.33. URL: http://dx.doi.org/10.1109/AP2PS.2009.33.

[40] Gyuwon Song, Suhyun Kim and Daeil Seo. 'Replica placement algorithm for highly available peer-to-peer storage systems'. In: *Advances in P2P Systems, 2009. AP2PS'09. First International Conference on*. IEEE. 2009, pp. 160–167.

[41] Angelos Stavrou, Dan Rubenstein and Sambit Sahu. 'A lightweight, robust p2p system to handle flash crowds'. In: *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on*. IEEE. 2002, pp. 226–235.

[42] Moritz Steiner, Taoufik En-Najjary and Ernst W Biersack. 'A global view of kad'. In: *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM. 2007, pp. 117–122.

[43] Jeremy Stribling. *Planetlab all pairs ping*. 2005.

[44] Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman and Michael Ward. 'Gryphon: An Information Flow Based Approach to Message Brokering'. In: *IN PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING*. 1998.

[45] Daniel Stutzbach and Reza Rejaie. 'Understanding churn in peer-to-peer networks'. In: *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM. 2006, pp. 189–202.

[46] Spyros Voulgaris. 'Epidemic-Based Self-Organization in Peer-to-Peer Systems'. PhD thesis. Amsterdam, The Netherlands: Vrije Universiteit, 2006.

[47] Spyros Voulgaris, Daniela Gavidia and Maarten van Steen. 'CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays'. In: *J. Network Syst. Manage.* (2005), pages.

[48] Spyros Voulgaris, Etienne Rivière, Anne-Marie Kermarrec and Maarten van Steen. 'Sub-2-Sub: Self-Organizing Content-Based Publish Subscribe for Dynamic Large Scale Collaborative Networks'. In: *Proceedings of the* 5^{th} *International Workshop on Peer-to-Peer Systems (IPTPS06)*. Feb. 2006.

[49] Spyros Voulgaris and Maarten Van Steen. 'Hybrid dissemination: adding determinism to probabilistic multicasting in large-scale P2P systems'. In: *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*. Springer-Verlag New York, Inc. 2007, pp. 389–409.

[50] Spyros Voulgaris, Maarten van Steen and Konrad Iwanicki. 'Proactive gossip-based management of semantic overlay networks'. In: *Concurr. Comput. : Pract. Exper.* 19.17 (2007), pp. 2299–2311. ISSN: 1532-0626.

[51] Bernard Wong and Saikat Guha. 'Quasar: A Probabilistic Publish-Subscribe System for Social Networks'. In: *Proceedings of the 7th International Workshop on Peer-to-Peer Systems (IPTPS)*. Tampa Bay, FL, Feb. 2008.

[52] Shelley Q Zhuang, Ben Y Zhao, Anthony D Joseph, Randy H Katz and John D Kubiatowicz. 'Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination'. In: *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*. ACM. 2001, pp. 11–20.